60484100

CONTROL DATA CORPORATION

CYBER INTERACTIVE DEBUG VERSION 1 GUIDE FOR USERS OF FORTRAN VERSION 5

CDC® OPERATING SYSTEMS:

NOS 1

NOS/BE 1

REVISION RECORD

Revision	Description
A (05/16/80)	Original release documenting CYBER Interactive Debug Version 1 and FORTRAN Version 5 at PSR level 512.
B (05/25/84)	Revised at PSR level 512 to document support of the CYBER 170 800 Series models and the CYBER 180 Computer Systems.
C (09/10/84)	Revised at PSR level 626 to incorporate the following changes: state default options for interactive jobs; remove references to Time Limit; define maximum length of command line; explain function of underscore.

REVISION LETTERS I, O, Q, AND X ARE NOT USED

Address comments concerning this manual to:

CONTROL DATA CORPORATION
Publications and Graphics Division
P. O. BOX 3492
SUNNYVALE, CALIFORNIA 94088-3492

or use Comment Sheet in the back of this manual

LIST OF EFFECTIVE PAGES

New features, as well as changes, deletions, and additions to information in this manual are indicated by bars in the margins or by a dot near the page number if the entire page is affected. A bar by the page number indicates pagination rather than content has changed.

Page	Revision
Front Cover	-
Title Page	-
ii	C
iii/iv	C
v	C
vi	A
vii	C
viii	C
1-1	A
1-2	A
2-1 thru 2-5	A
3-1 thru 3-10	A
3-11	C
3-12 thru 3-23	A
3-24	C
3-25	A
3-26	A
3-27	C
3-28 thru 3-40	A
4-1	C
4-2 thru 4-23	A
5-1 thru 5-4	A
A-1	A
A-2	A
B-1	A
B-2	A
C-1 thru C-3	A
D-1 thru D-3	A
E-1	A
E-2	A
Index-1	A
Index-2	A
Comment Sheet/Mailer	C
Back Cover	-

PREFACE

This manual provides the FORTRAN programmer with assistance in the debugging of FORTRAN Version 5 programs under the control of the CDC® CYBER Interactive Debug Facility.

CYBER Interactive Debug (CID) operates under the following operating systems:

NOS/BE 1 for the CONTROL DATA® CYBER 180 Series; CYBER 170 Series; CYBER 70 Models 71, 72, 73, 74; and 6000 Series Computer Systems.

NOS 1 for the CDC CYBER 180 Series; CYBER 170 Series; CYBER 70 Models 71, 72, 73, 74; and 6000 Series Computer Systems.

You should have a copy of the CYBER Interactive Debug reference manual available for reference, but you need not be familiar with the manual. In addition, you should be familiar with FORTRAN 5 and should be able to run jobs interactively under either NOS/BE INTERCOM or the NOS Interactive Facility.

This guide provides a tutorial approach to CID beginning with basic features and proceeding through more advanced features. Section 1 provides some background information and presents a summary of the features of CID. Section 2 describes the method for initiating a debug session with CID, and describes several useful CID commands; this section contains sufficient information to allow the casual user to make productive use of CID. Sections 3 through 5 describe features which are helpful in debugging more complex programs. This guide is not comprehensive in its approach to CID; only those features considered useful to FORTRAN programmers are described. Most of the features described in this manual are illustrated by actual examples of debug sessions. This is intended to help you become familiar with CID notational conventions and with information produced by CID.

The Software Publications Release History serves as a guide in determining which revision level of software documentation corresponds to the Programming System Report (PSR) level of installed site software.

Additional information can be found in the listed publications.

The following publications are of primary interest:

Publication	Publication Number
CYBER Interactive Debug Version 1 Guide for User's of FORTRAN Version 5 Online	L60484100
CYBER Interactive Debug Version 1 Reference Manual	60481400
CYBER Interactive Debug Version 1 Reference Manual Online	L60481400
FORTRAN Version 5 Reference Manual	60481300
FORTRAN Version 5 Reference Manual Online	L60481300

The following publications are of secondary interest:

Publication	Publication Number
CYBER Loader Version 1 Reference Manual	60429800
INTERCOM Version 5 Reference Manual	60455010
NOS Version 1 Manual Abstracts	84000420
NOS/BE Version 1 Manual Abstracts	84000470
Network Products Interactive Facility Version 1 Reference Manual	60455250

Publication	Publication Number
Network Products Interactive Facility Version 1 User's Guide	60455260
Software Publications Release History	60481000
XEDIT Version 3 Reference Manual	60455730

CDC manuals can be ordered from Control Data Corporation, Literature and Distribution Services, 308 North Dale Street, St. Paul, Minnesota 55103.

This manual describes a subset of the features and parameters documented in the CYBER Interactive Debug Version 1 Reference Manual and the FORTRAN Version 5 Reference Manual. Control Data cannot be responsible for the proper functioning of any features or parameters not documented in the CYBER Interactive Debug Version 1 Reference Manual.

CONTENTS

INDEX

FIGURES

TABLES

INTRODUCTION

The CYBER Interactive Debug facility (CID) allows the FORTRAN programmer to interactively debug an executing object program. CID can be used with FORTRAN 5 programs compiled under the NOS or NOS/BE operating systems.

Use of CID requires a mode of execution called debug mode. Debug mode is established by a control statement. As long as debug mode is in effect, execution of all user programs takes place under control of CID. CID, in turn, allows you to enter commands that perform the following operations:

- Suspend program execution at specified locations.
- Suspend program execution on the occurrence of selected conditions, such as modification of a variable.
- Display the values of variables, arrays, and common blocks while execution is suspended.
- Change the values of variables, arrays, or common blocks within the program while execution is suspended.
- Resume program execution at the location where it was suspended, or at another location.

WHAT IS INTERACTIVE DEBUGGING?

Interactive debugging means that you debug your program while it is executing. In interactive mode, CID allows you to suspend execution of your program and enter commands directly from a terminal while execution is suspended. CID executes each command immediately after it is entered. Program execution remains suspended until resumed by the appropriate command. In this manner, you can control and monitor the execution of your program, stopping at desired points to examine and modify the values of program variables.

SPECIAL CID FEATURES FOR FORTRAN PROGRAMS

CID provides certain features currently available only to FORTRAN 5 programs compiled in debug mode. These features include commands with a FORTRAN-like syntax and the capability of symbolically referencing locations within an object program. The commands available only in debug mode are indicated in appendix E.

For purposes of this user's guide, it is assumed that FORTRAN programs to be executed under CID control are compiled in debug mode; therefore, in the discussions of the CID capabilities, no distinction is made between standard CID features and the special features available to FORTRAN programs. It is possible, though more difficult, to use CID with programs not compiled in debug mode. Refer to the CYBER Interactive Debug reference manual for a description of this capability.

WHY USE CID?

Conventional debugging techniques often require the use of load maps, object listings, and octal dumps. In addition, it is often necessary to recompile a FORTRAN program several times to make corrections or to add statements that print intermediate values of program variables. These debugging techniques can be expensive in terms of both machine time and programmer time.

CID, however, does not require a knowledge of assembly language or the ability to interpret memory dumps. You can completely debug a program with CID by referring only to a source listing and by referencing variables and line numbers symbolically. In many cases, a FORTRAN program need be compiled only once; the resulting object program can be executed repeatedly with different CID commands specified for each run. Since CID allows you to make changes to your program's data and flow of control as execution proceeds, you can often accomplish, in a single session, debugging that would normally require several compilations. Thus, considerable time can be saved, especially when you are debugging programs that are time-consuming to compile or execute.

A disadvantage of CID is that OPT=0 compilation mode is required if the special FORTRAN commands and symbolic capabilities are to be used. Since a program that executes correctly when compiled with OPT=0 might not do so when compiled in a higher mode of optimization, a program debugged with OPT=0 should also be tested with OPT=1 or OPT=2. If the program does not execute correctly in a higher mode of optimization, you must either use conventional debugging techniques or use CID without the symbolic referencing capabilities.

PROGRAMMING FOR EASE OF DEBUGGING

When coding a FORTRAN program, there are certain guidelines you can follow to make debugging easier. An important consideration is program modularity. Simply stated, program modularity means limiting the size of program units and dividing programs into subprograms that perform logical functions. A modular program is easier to understand, easier to modify, and easier to debug.

CID lends itself to use with a modular program. Through CID, you can gain control on entry into and on exit from a subprogram. You can use CID to display values passed to a subprogram, intermediate values used in computations within the subprogram, and values output from the subprogram. By specifying special parameters on CID commands, you can restrict the scope of the commands to particular program units.

Using a style of coding that avoids GO TOs and minimizes branches can be an aid in the debugging process. A program that contains a minimum of branches and flows logically from top to bottom is much easier to understand

than one that contains many unnecessary branches. The block IF structures of FORTRAN 5 facilitate a structured style of programming, and should be used wherever possible to simplify program flow. CID provides features that allow you to trace the flow of control of your executing program; this process is much easier if the program avoids needlessly complex logic.

You should avoid programming tricks and shortcuts, particularly if they depend on system idiosyncracies. For example, although some systems initialize memory to zero, it is best to include statements in your program which perform all appropriate initialization.

CID should not be considered a substitute for proper programming practices. Even though CID offers many powerful features, a well-written program is much easier to debug.

Program carefully and try to minimize the number of errors. Performing a careful visual scan of the program before execution can reveal many of the more obvious errors. Use other debugging aids, such as the FORTRAN cross reference listing and Post Mortem Dump. It is better to have correct code to begin with than to spend time debugging.

WHAT EFFECT DOES CID HAVE ON PROGRAM SIZE AND EXECUTION TIME?

If the special FORTRAN features are to be used in a debug session, the program must be compiled in debug mode. This requires OPT=0 compilation mode. OPT=0 compilation generates unoptimized object code, generally resulting in faster compilation, but slower execution. The minimum field length requirement for a program compiled in OPT=0 mode is 40000_8 words. In addition, compiling in debug mode generates additional code for use by CID.

The CID module, which is loaded into the user's field length, increases the memory requirement by approximately 4000 words; the minimum field length requirement is 53000 words. Programs that become excessively large should be modularized, and the modules debugged separately.

Certain CID features require a mode of execution called interpret mode (described in section 3) which requires much more execution time than normal execution. This can be a significant problem in some programs. In most cases, however, you can substitute an alternate feature that does not require interpret mode.

OVERLAY DEBUGGING

CID can be used with programs containing overlays. CID provides features intended specifically for the debugging of programs with overlays, including a special trap that allows you to suspend program execution when an overlay is loaded. Overlay debugging is described in section 5.

CID cannot be used with programs loaded by either SEGLOAD or the user-call loader.

BATCH MODE DEBUGGING

Although CID is intended to be used interactively, it can be used in batch mode. Batch mode debugging is described in appendix D.

GETTING STARTED 2

This section summarizes the operations necessary for conducting a debug session and introduces several CYBER Interactive Debug (CID) notation conventions. At the end of the section, several basic commands are presented and used in a sample session. These commands enable you to conduct a simple but useful debug session.

ENTERING THE CID ENVIRONMENT

To execute a program under CID control (and to make use of the FORTRAN capabilities), you must compile and execute the program in debug mode. Debug mode is turned on by a system control statement.

DEBUG CONTROL STATEMENT

The DEBUG control statement activates debug mode. The format of this statement is:

    DEBUG
or
    DEBUG(ON)

When a FORTRAN program is compiled in debug mode, special symbol tables for use by CID are generated as part of the object code. When the program is subsequently executed in debug mode, all of the CID features can be used. Note that a program that has not been compiled in debug mode can still be executed in debug mode, but program locations cannot be referenced symbolically. (This precludes the use of same of the features described in section 3.)

When debug mode is on, you can interact with the operating system and perform all other terminal activities in a normal manner; only FORTRAN compilations and relocatable loads are affected.

If you are using the FORTRAN subsystem (NOS) or the INTERCOM EDITOR (NOS/BE), you can compile and execute in debug mode by specifying the DEBUG control statement before entering the RUN command.

The statement to deactivate debug mode is:

    DEBUG(OFF)

When debug mode is off, programs that were compiled in debug mode execute normally. It is necessary to enter this statement only if you do not wish subsequent compilations or executions to occur under CID control.

EXECUTING UNDER CID CONTROL

A debug session consists of the sequence of interactions between you and CID which takes place while your object program is executing in debug mode. The session begins when you initiate execution of your object program and ends when you enter the QUIT command.

If you are executing under the NOS/BE EDITOR or the NOS FORTRAN subsystem, you can begin the session by issuing the appropriate RUN command, since this command automatically initiates program execution after compilation is complete. If you are compiling with an FTN5 control statement, the session is initiated by specifying the GO parameter or by entering the name of the binary object file (default name is LGO) after the compilation has completed. The system loads the CID program module, your binary program, and system and library modules. Control then transfers to an entry point in CID. CID then issues the message:

    CYBER INTERACTIVE DEBUG
    ?

The ? character is a prompt signifying that CID is waiting for user input. At this point you can enter CID commands.

The examples in figure 2-1 show the statements necessary for compiling a program and initiating a debug session, under the NOS and NOS/BE operating systems.

Debugging a program can require more than one debug session. If this is the case, you can terminate the current session and initiate a new session. Note that once a program has been compiled in debug mode, it is not necessary to recompile in order to conduct another debug session with the same program. You can initiate another session merely by entering the binary file name (the normal method of executing a program).

ENTERING CID COMMANDS

The CID prompt for user response is a question mark (?). In response to the ? character, enter a CID command and press the transmission key (RETURN on most terminals). CID then processes the command, issues an informative message indicating the disposition of the command or displays any output that the command calls for, and issues another ? prompt. CID continues to issue prompts after processing commands until you enter the command to resume execution of your program, or until you terminate the session.

If you enter a command incorrectly, CID displays a diagnostic message. One such message is:

    ERROR- UNKNOWN COMMAND

If this message appears, determine the correct format, and reenter the command. You can use the HELP command, described later in this section for assistance with command formats.

Example 1:

```
COMMAND- editor                          Enter edit mode.

..edit,proga,seq                         Make PROGA the edit file.

..debug                                  Turn on debug mode.

..run,ftn5                               Compile program and initiate debug session.

      57500 CM STORAGE USED.
      0.064 CP SECONDS COMPILATION TIME.

CYBER INTERACTIVE DEBUG
?
```

Example 2:

```
COMMAND- debug                           Turn on debug mode.

COMMAND- ftn5,i=proga,l=list             Compile program.

      57500 CM STORAGE USED.
      0.061 CP SECONDS COMPILATION TIME.
COMMAND- lgo                             Initiate debug session.

CYBER INTERACTIVE DEBUG
?
```

Example 3:

```
/fortran                                 Enter FORTRAN subsystem.
OLD, NEW, OR LIB FILE: old,proga         Designate PROGA as primary file.

 READY.
debug                                    Turn on debug mode.

 READY.
run                                      Compile program and initiate debug session.

 79/11/02. 08.30.28.
PROGRAM   PROGA

 CYBER INTERACTIVE DEBUG
?
```

Figure 2-1. Inititating a Debug Session

SHORTHAND NOTATION FOR CID COMMANDS

Most standard CID commands have a shorthand form that permits you to omit the comma separator and to substitute abbreviations for the command name and certain parameters. For example, the command:

SET,TRAP,LINE,*

can be expressed as:

STL*

The shorthand notation provides a more convenient method of specifying commands; you are encouraged to use this form as you become more familiar with CID. However, for purposes of clarity and consistency, only the full command forms are used in this manual. The short command forms are listed in appendix E.

REFERENCING SOURCE STATEMENTS

Many of the CID command formats require you to indicate a specific statement within the program you are debugging. Source statements are referenced either by line sequence number or by statement label by using the following notation.

LINE NUMBER SPECIFICATION

The notation for specifying a sequence number is:

L.n

where n is the statement sequence number or the number indicated on the compiler-generated source listing for programs without sequence numbers. This notation denotes

the source line having the specified sequence number. Leading zeros can be omitted. Some examples of sequence number references are as follows:

L.130
L.1
L.26

STATEMENT LABEL SPECIFICATION

You can also reference a program statement by specifying a statement label assigned to that statement in the source program. A statement label specification has the form:

S.n

where n is the statement label. Only executable statements can be referenced in this manner. For example:

S.10

designates the source statement having the label 10.

SOME ESSENTIAL COMMANDS

The following paragraphs describe several CID commands that enable you to conduct simple debug sessions. These are the GO command, the QUIT command, the PRINT command, and the SET,BREAKPOINT command. The HELP command, which provides a quick summary of information about various CID subjects, is also described. (These commands are described in greater detail in section 3.) The command forms presented here allow you to debug programs consisting of a single program unit only. To debug programs containing multiple program units (main program, subroutines, and function subprograms), you must be familiar with the home program concept described in section 3.

GO

The command to initiate or resume program execution is:

GO

If entered at the beginning of the debug session, this command initiates program execution. If entered after execution has been suspended, this command causes execution to resume at the statement where it was suspended.

Once execution of your program has been suspended, any number of CID commands can be entered. Execution remains suspended until you enter GO.

QUIT

The command to terminate a debug session is:

QUIT

In response to the QUIT command, CID displays the following message under NOS/BE and the NOS batch subsystem:

DEBUG TERMINATED

Under NOS FORTRAN subsystem:

SRU n.nnn UNTS

RUN COMPLETE

The QUIT command causes an exit from the current session and a return to system command mode. Files accessed by the FORTRAN program are closed. Note, however, that debug mode remains on until DEBUG(OFF) is specified. You can initiate another debug session for the same program, without recompiling, by entering the binary file name (as described under Initiating a Debug Session).

Traps, breakpoints, and other alterations to the object program exist only for the duration of the debug session. When the session is terminated, any changes made to the program are lost, and the program reverts to its compiled version. You can terminate a debug session any time you have control (CID has issued a ? prompt). The object program can then be executed normally, or it can be executed again under CID control.

PRINT

CID provides several commands for displaying the values of program variables. The simplest of these is the command:

PRINT*,list

where list is a list of program variables. This command has the same format as the FORTRAN list-directed PRINT statement. The list can contain expressions and implied DO loops.

This command lists the values of the specified program variables. Values are formatted according to type declared, implicitly or explicitly, in the source program (integer, real, logical, character, boolean, or complex).

Examples:

PRINT*,X1,X2,A+B

PRINT*,'VALUES',B(1),B(2),B(3)

PRINT*, (ARR(I),I=1,10)

SET,BREAKPOINT

A breakpoint is a location within a program where execution is to be suspended. The command to establish a breakpoint has the form:

SET,BREAKPOINT,loc

where loc is a line number specification (L.n) or statement label specification (S.n) as described under Referencing Source Statements. When the specified statement is reached in the flow of execution, control transfers to CID which then allows you to enter CID commands. Typically, commands are entered to examine the values of program variables, and execution is resumed.

Examples:

SET,BREAKPOINT,L.14

Sets a breakpoint at line 14.

SET BREAKPOINT,S.50

Sets a breakpoint at the statement labeled 50.

You can establish breakpoints at any time in the debug session when execution is suspended and CID has issued a ? prompt.

A breakpoint can be established at any executable statement. Only one breakpoint can be set at a single statement; however, a breakpoint can be set at a location where a trap occurs and both the trap and the breakpoint are recognized. Breakpoints are always recognized first.

Establishing a breakpoint at a specified location does not alter execution of the statement at that location. When a breakpoint is encountered during execution, CID gains control before the statement is executed. When execution is resumed, execution begins with the statement at the breakpoint location.

When a breakpoint is encountered, CID receives control and issues the following message:

*B #n AT loc

where n is a breakpoint number assigned by CID, and loc is the statement (S.n or L.n) where the breakpoint was set. Breakpoints are assigned consecutive numbers in the order they are established, beginning with 1.

HELP

CID provides a HELP command that displays a brief summary of information about specific CID subjects and commands. You can issue the HELP command whenever you need assistance with a particular aspect of CID.

Simply entering the command:

HELP

causes CID to display a list of subjects. To obtain additional information about any subject in the list, enter:

HELP,subject

For example, the command HELP,ERROR displays a brief description of error processing.

A useful form of the HELP command is HELP,CMDS which displays a complete list of CID commands and a brief explanation of each. You can obtain a more detailed explanation of any CID command by entering:

HELP,command

where command is any CID command. The HELP command does not provide the same level of detail as the CID reference manual, however, and should not be considered a substitute for the reference manual.

The HELP command is illustrated in figure 2-2, which shows the entry of the command HELP,SET,BREAKPOINT to display a summary of the command parameters.

SUMMARY

A significant characteristic of CID is that much of its power exists in a few commands. It is not necessary to have a complete knowledge of all the CID commands, to take advantage of the most powerful features of CID.

Following is a step-by-step summary of information presented in this section.

To use CID:

1. Type DEBUG to turn on debug mode.

2. Compile and load your program in a normal manner. Control transfers to CID when execution begins. CID displays a message at the terminal and waits for your input.

3. Set breakpoints as desired.

   To set a breakpoint at a line number or statement label enter:

   SET,BREAKPOINT,L.n
   SET,BREAKPOINT,S.n

   where n is a line number or statement label.

4. Enter GO to begin execution of your program.

   CID executes your program in a normal manner, but returns control to you when a breakpoint occurs.

```
? help,set,breakpoint
 SB  -  SET  BREAKPOINT - ALLOWS YOU TO SET A BREAKPOINT AT A
 SPECIFIC LOCATIONS IN USER'S PROGRAM.  THE FORM OF  THE  SET
 BREAKPOINT COMMAND IS.
      SB <LOCATION>,<FIRST>,<LAST>,<STEP>
 WHERE <LOCATION> IS THE LOCATION IN YOUR PROGRAM AT WHICH
 YOU WANT THE BREAKPOINT SET.
 <FIRST>, <LAST> AND <STEP> ARE OPTIONAL AND ARE DEFAULTED TO
 1, 131071 AND 1 RESPECTIVELY.  THE BREAKPOINT IS NOT HONORED
 UNTIL <LOCATION> HAS BEEN HIT <FIRST> TIMES. BUT, IT WILL BE
 HONORED WHEN <LOCATION> IS HIT THE <FIRST>TH TIME AND EACH
 <STEP>TH TIME AFTER THAT AS LONG AS <LAST> IS NOT EXCEEDED.
 IF YOU TERMINATE THE SB COMMAND WITH AN OPEN BRACKET [, THEN
 ALL COMMANDS UP TO A CLOSE BRACKET ] WILL BE COLLECTED  SUCH
 THAT  WHEN THE BREAKPOINT IS HONORED, THOSE COMMANDS WILL BE
 EXECUTED.
?
```

Figure 2-2. Example of HELP Command

5. At this point, you can display the values of program variables with the statement:

   PRINT*,variable list

   To resume execution, enter GO.

6. Enter QUIT to terminate the session. Enter DEBUG(OFF) to turn off debug mode.

Debug sessions can become complicated. Always try to keep debug sessions short and simple. If necessary, correct known bugs, recompile your program, and conduct additional debug sessions.

SAMPLE DEBUG SESSION

The preceding commands are now used to conduct a simple debug session. As you study the examples in this guide, keep in mind that these examples are intended to illustrate the various CID features; they are not intended to present a suggested sequence of commands for debugging all programs. The actual commands used in a given debug session depend on the program in question and, often, on the inclination of the programmer.

A FORTRAN program and a debug session log are illustrated in figure 2-3. The program calculates the area of a triangle given the coordinates of the endpoints of the three sides. After the session is initiated, a breakpoint is set at line 6. When line 6 is reached during execution, CID obtains control, and prompts for user input. The PRINT command is entered to display the intermediate values used in the area calculation. Execution is then resumed, and the program runs to completion. (The END trap, which occurs on normal program termination, is described in section 3.) The final result is displayed, and the session is terminated.

```
                    PROGRAM ATR1          74/74    OPT=0

          1                     PROGRAM ATR1
          2                     DATA X1,Y1,X2,Y2,X3,Y3 /0.0,0.0,0.0,2.0,2.0,0./
          3                     S1=SQRT((X2-X1)**2 + (Y2-Y1)**2)
          4                     S2=SQRT((X3-X1)**2 + (Y3-Y1)**2)
          5                     S3=SQRT((X3-X2)**2 + (Y3-Y2)**2)
          6                     T=(S1+S2+S3)/2.0
          7                     A=SQRT(T*(T-S1)*(T-S2)*(T-S3))
          8                     END

CYBER INTERACTIVE DEBUG
?set,breakpoint,L.6                     Set breakpoint at line 6.

?go                                     Initiate program execution.

*B #1, AT L.6                           Execution suspended at line 6.
?print*,s1,s2,s3                        Display values of intermediate variables.

 2. 2. 2.828427124746
?go                                     Resume execution.

*T #17, END IN L.8                      Trap message signifies program has terminated.
?
     END  ATR1                      }
     12400 MAXIMUM EXECUTION FL.    }   FORTRAN messages.
      .168 CP SECONDS EXECUTION TIME.}
print*,a                                Display final result.

 2.
?quit                                   Terminate debug session.

  DEBUG TERMINATED
```

Figure 2-3. Program ATR1 and Debug Session

Once you have compiled your FORTRAN program in debug mode and initiated a debug session you are ready to begin interactive debugging. Program execution under CYBER Interactive Debug (CID) control involves an interaction between you and CID; you specify conditions for which program execution is to be suspended, and CID gives control to you when these conditions are satisfied and allows you to enter various CID commands to examine and alter the status of the program.

The preceding section presented some elementary commands that can be used to conduct a simple debug session. This section begins with a discussion of the home program, a concept you should be famlliar with in order to debug programs containing subroutines. Following this discussion is additional information on the commands presented in section 2 and descriptions of some other commands and CID features that allow you to make more productive use of CID. The commands discussed in this section enable you to:

- Suspend program execution; commands are SET,BREAKPOINT and SET,TRAP.
- Display the current values of program variables and arrays at the terminal while execution is suspended; commands are PRINT, DISPLAY, and LIST,VALUES.
- Alter the contents of variables and arrays; commands are MOVE and assignment.

HOME PROGRAM

FORTRAN programs consist of a main program and, optionally, one or more subprograms. Variable names within a program are local to the program unit in which they are defined; that is, variable names are known only within the program in which they are used. This concept is illustrated in figure 3-1. In this example, the variable A is defined twice: once in the main program and once in the subroutine. However, execution of the statement A=1.0 in the subroutine does not alter the contents of the variable A in the main program; the value printed for A is always 1.0. Although two variables have the same name, each variable is local to the program unit in which it is defined.

The same concept of locality applies to CID. When a program consisting of multiple program units is executed under CID control, execution can be suspended in the main program or in any of the subprograms. The default home program is defined as the program unit in control at the time of suspension. Variable names, line numbers, and statement labels specified in CID commands are those contained in the home program.

The home program concept is illustrated by the debug session in figure 3-2 that is produced by executing the program in figure 3-1 in debug mode. Breakpoints are set to suspend execution in the main program after the call to SUBA, and in SUBA itself. When execution is suspended in SUBA, SUBA is the home program and the PRINT command shows 2.0 as the value of A; when execution is suspended in the main program, the value of A is 1.0.

```
                PROGRAM MAIN        74/74    OPT=0

          1              PROGRAM MAIN
          2              A=1.0
          3              CALL SUBA
          4              PRINT*, 'A = ',A
          5              END

            SUBROUTINE SUBA        74/74    OPT=0

          1              SUBROUTINE SUBA
          2              A=2.0
          3              RETURN
          4              END

 A = 1. <------------------------------------- Program Output.
        END  MAIN
        12400 MAXIMUM EXECUTION FL.
         .056 CP SECONDS EXECUTION TIME.
```

Figure 3-1. Main Program and Subroutine Illustrating Local Variables

```
CYBER INTERACTIVE DEBUG
?set,breakpoint,l.4 ◄────────── Set a breakpoint at line 4 of main program.

?set,breakpoint,p.suba_l.3◄── Set a breakpoint at line 3 of subroutine SUBA.

?go ◄────────────────────────── Initiate execution.

*B #2, AT P.SUBA_L.3 ◄───────── Execution suspended at line 3 of SUBA.
?print*,a◄───────────────────── Display value of A defined in SUBA.

 2.
?go ◄────────────────────────── Resume execution.

*B #1, AT L.4  (OF P.MAIN)◄── Execution suspended at line 4 of main program.
?print*,a◄───────────────────── Display value of A defined in main program.

 1.
?quit

  DEBUG TERMINATED
```

Figure 3-2. Debug Session Illustrating Home Program Concept

REFERENCING LOCATIONS OUTSIDE THE HOME PROGRAM

In some cases, you might wish to reference a location in a program unit other than the home program. For example, when execution is suspended in the main program, you might want to set a breakpoint or display a value local to another subprogram. To accomplish this, you can do either of the following:

- Use CID commands which allow a variable name, line number, or statement label specification to be qualified by a program unit name.
- Designate a new home program with the SET,HOME command.

QUALIFICATION NOTATION

Qualification notation allows you to specify a variable, line number, or statement label occurring in a program unit other than the home program. This notation has the following forms:

P.prog_var

Denotes variable var in program unit prog.

P.prog_L.n

Denotes line n in program unit prog.

P.prog_S.n

Denotes statement labeled n in program unit prog.

The program unit name and variable, line number, or statement label specification are separated by an underscore character. Qualification notation is valid for all CID commands except the PRINT and assignment commands described later in this section.

Examples:

P.NEWT_X

Variable X in program unit NEWT.

P.SUBZ_L.410

Line 410 in program unit SUBZ.

P.FUNC_S.12

Statement labeled 12 in program unit FUNC.

Qualification notation can be substituted for the normal variable, line number, or statement label specification in CID commands for which this notation is valid, as in the following example:

SET,BREAKPOINT,P.CAT_L.25

Sets a breakpoint at line 25 of program unit CAT.

Qualification notation also appears in many types of CID informative output. For example, the message:

*B #1 AT P.XYZ_L.14

indicates that a breakpoint was encountered at line 14 of program unit XYZ.

CID notation forms are summarized in table 3-1.

SET,HOME COMMAND

As an alternative to the qualification notation or in cases where this notation is invalid, you can specify locations outside the default home program by first issuing the command:

SET,HOME,P.prog

where prog is a program unit name. This command changes the home program. Any unqualified variable names, line numbers, or statement labels specified after entering the SET,HOME command belong to prog. It is important to note that the SET,HOME command does not alter the location where execution resumes when you issue GO; execution always resumes at the location where it was suspended regardless of SET,HOME specification. In addition, when execution is resumed, a previous SET,HOME specification is lost, and the home program reverts to the one currently executing.

TABLE 3-1. CID NOTATION

Notation	Description
P.prog	Program unit prog.
var	Simple or subscripted variable name.
P.prog_var	Variable in program unit prog.
L.n	Source line having sequence number n.
P.prog_L.n	Source line having sequence number n in program unit prog.
S.n	Source statement labeled n.
P.prog_S.n	Source statement labeled n in program unit prog.
C.	Unlabeled common block.
C.blk	Common block blk.
XC.blk	Common block blk in extended memory.
C.blk_n	Word n+1 of common block blk.

The following two examples produce the same results: a breakpoint is set in program unit XXX while execution is suspended in program unit YYY. Program execution is then resumed at the point of suspension (line 21 of program unit YYY).

Example 1:

```
  #B #1 AT P.YYY_L.21
? SET,BREAKPOINT,P.XXX_L.5
? GO
```

Example 2:

```
  #B #1 AT P.YYY_L.21
? SET,HOME,P.XXX
? SET,BREAKPOINT,L.5
? GO
```

The debug session in figure 3-3, produced by executing the program in figure 3-1 in debug mode, illustrates another example of the SET,HOME command. Note that on program termination, the home program is once again the main program. To print the value of A in subroutine SUBX, a SET,HOME command must be entered.

DEBUGGING AIDS FOR PROGRAMS WITH MULTIPLE PROGRAM UNITS

CID provides features that can be helpful when your program contains a number of subroutine calls. The #HOME debug variable tells you the name of the program unit where execution is suspended (unless you changed it with the SET,HOME command). The TRACEBACK command is useful for programs containing a number of calls to a given subprogram; when execution is suspended in a subprogram, it tells you from where the subprogram was called.

#HOME DEBUG VARIABLE

The #HOME debug variable is a special variable belonging to CID. This variable always contains the name of the current home program. You can display the contents of this variable with the command:

```
DISPLAY,#HOME
```

```
CYBER INTERACTIVE DEBUG
?set,breakpoint,l.4 <---------- Set a breakpoint at line 4 of home program (MAIN).

?go <---------- Initiate execution.

*B #1, AT L.4 <---------- Execution suspended at line 4.
?print*,a <---------- Display value of A defined in home program (MAIN).

 1.
?set,home,p.suba <---------- Designate subroutine SUBA as home program.

?print*,a <---------- Display value of A defined in home program (SUBA).

 2.
?go <---------- Resume execution (at point of suspension in MAIN).

A = 1.
*T #17, END IN P.MAIN_L.5 <---------- Program terminates.
?
      END  MAIN
      15700 MAXIMUM EXECUTION FL.
       .260 CP SECONDS EXECUTION TIME.
print*,a <---------- Display value of A defined in home program (MAIN).

 1.
?
```

Figure 3-3. Debug Session Illustrating SET,HOME Command

CID displays the name of the current home program in the form:

#HOME=P.prog

This variable is useful for determining the subroutine where execution is suspended, although CID normally displays the home program name when suspension occurs. Note, however, that if you change the home program with the SET,HOME command, #HOME contains the name of the new home program.

Debug variables are described in greater detail later in this section (under Debug Variables).

TRACEBACK COMMAND

The TRACEBACK command displays a list of subroutine levels from the level of the current home program through the main level. At each level, TRACEBACK displays the name of the program unit that last called the current subroutine, and the line number within the program unit where the call occurred. The format of this output is P.name_L.n. The form of the TRACEBACK command is:

TRACEBACK

> Displays a traceback list beginning at the current home program.

The TRACEBACK command is illustrated by the program and debug session shown in figure 3-4. The program contains three levels of subroutine calls. Breakpoints are set in subroutines SUB3 and SUB4, the lowest level. When execution is suspended at these breakpoints, the TRACEBACK command is entered.

ERROR AND WARNING PROCESSING

Each time you enter a command, CID checks the command for correctness. If errors are detected, CID issues either an error or a warning message.

```
00100 PROGRAM MAIN
00110 CALL SUB1
00120 END
00130C
00140 SUBROUTINE SUB1
00150 CALL SUB2
00160 RETURN
00170 END
00180C
00190 SUBROUTINE SUB2
00200 CALL SUB3
00210 CALL SUB4
00220 RETURN
00230 END
00240C
00250 SUBROUTINE SUB3
00260 RETURN
00270 END
00280C
00290 SUBROUTINE SUB4
00300 RETURN
00310 END


 CYBER INTERACTIVE DEBUG
? set,breakpoint,p.sub3_l.260
? set,breakpoint,p.sub4_l.300
? go
 *B #1, AT P.SUB3_L.260 ←-------------- Execution suspended in SUB3.
? traceback ←-------------------------- Initiate traceback from SUB3.
 P.SUB3 CALLED FROM P.SUB2_L.200
 P.SUB2 CALLED FROM P.SUB1_L.150
 P.SUB1 CALLED FROM P.MAIN_L.110
? traceback,p.sub4 ←------------------- Attempt to initiate traceback from SUB4.
 *ERROR - PROGRAM SUB4 NOT CALLED
? go
 *B #2, AT P.SUB4_L.300 ←-------------- Execution suspended in SUB4.
? traceback ←-------------------------- Initiate traceback from SUB4.
 P.SUB4 CALLED FROM P.SUB2_L.210
 P.SUB2 CALLED FROM P.SUB1_L.150
 P.SUB1 CALLED FROM P.MAIN_L.110
? traceback,p.sub3 ←------------------- Initiate traceback from SUB3.
 P.SUB3 CALLED FROM P.SUB2_L.200
 P.SUB2 CALLED FROM P.SUB1_L.150
 P.SUB1 CALLED FROM P.MAIN_L.110
?
```

Figure 3-4. Program and Debug Session Illustrating TRACEBACK Command

ERROR MESSAGES

CID issues an error message whenever it encounters a command that cannot be executed. Error messages are usually caused by a misspelled command or an illegal or misspelled parameter. CID does not attempt to execute an erroneous command. CID error messages, which are followed by a user prompt, have the form:

```
*ERROR-text
?
```

The text contains a brief description of the error.

In response to an error message, you should consult the CID reference manual or use the HELP command to determine the correct command form, and reenter the command. Figure 3-5 illustrates some typical error messages. The first message is caused by a misspelled PRINT command. In the second example, the command is syntactically correct but the home program does not contain a statement labeled 10; a qualifier directing CID to another program unit was omitted from the statement label specification.

```
 *B #1, AT L.120
? pirnt*,a
 *ERROR - UNKNOWN COMMAND
? print*,a
  1.
? set,breakpoint,s.10
 *ERROR - NO EXECUTABLE STATMENT 10
? set,breakpoint,p.sub_s.10
?
```

Figure 3-5. Debug Session Illustrating Error Messages

WARNING MESSAGES

CID issues a warning message if a command you have entered will have consequences you might not be aware of or if the command will result in CID action other than that which you have specified. The warning message is followed by a special input prompt; in response to this prompt, you can tell CID either to execute the command or to ignore it. The format of a warning message is:

```
*WARN-message
OK?
```

The message describes the action CID will take if allowed to execute the command. In response to a warning message you can enter the following:

YES or OK	CID executes the command.
NO	CID disregards the command.
Any CID Command	CID disregards the previous command and executes the new one.

Some examples of warning messages are illustrated in figure 3-6. The first message is generated wnen the programmer attempts to set a breakpoint beyond the last executable statement of the home program. In this case, the programmer omitted a qualifier from the line number specification in the SET,BREAKPOINT command. The correct command is entered in response to the OK? prompt. The second message occurs after a CLEAR,TRAP command is entered. CID warns that this command removes all existing traps, and allows the programmer to reconsider. The programmer then enters an affirmative response, and CID executes the CLEAR,TRAP command.

Warning messages can be suppressed by an option on the SET,OUTPUT command, described later in this section (under Control of CID Output). In this case, CID automatically takes the action indicated in the message, without providing notification.

Refer to the CID reference manual for a complete list of warning messages and an explanation of each.

TRAPS AND BREAKPOINTS

When conducting a debug session, you must initially provide for gaining interactive control at some point within your program. CID provides two methods of doing this: traps and breakpoints.

A breakpoint (introduced in section 2) causes program execution to be suspended when a specified statement is reached in the flow of execution. A trap causes execution to be suspended when a specified condition is detected during execution. Both traps and breakpoints cause CID to give control to you so that you can examine and alter the status of your program at various points during execution.

In a typical debug session, you establish traps and breakpoints prior to initiating execution of the program. When a trap condition occurs or a breakpoint is detected during execution, CID receives control and, in turn, gives you the opportunity to enter CID commands.

```
? set,breakpoint,l.190
 *WARN - LINE 190 NOT EXECUTABLE - LINE 170 WILL BE USED
 OK ? set,breakpoint,p.getr_l.190
? clear,trap
 *WARN - ALL WILL BE CLEARED
 OK ? ok
?
```

Figure 3-6. Debug Session Illustrating Warning Messages

In most debugging situations, breakpoints, rather than traps, are recommended for suspending execution. Traps can be useful in certain cases, but some trap types require you to be familiar with COMPASS instructions; only trap types useful to most FORTRAN programmers are covered here. Breakpoints allow you to suspend execution at any executable statement in your program and can, in most cases, be substituted for traps.

Traps and breakpoints exist only for the duration of a debug session. Once a session is terminated, all traps and breakpoints set during a session cease to exist. An object program is not permanently altered by any traps or breakpoints established during a session.

CID provides commands that enable you to:

- Establish traps and breakpoints.
- Display a list of existing traps and breakpoints.
- Remove existing traps and breakpoints.
- Save trap and breakpoint definitions on a separate file for use in a later debug session.

SUSPENDING EXECUTION WITH BREAKPOINTS

A breakpoint is a mechanism established at a specified location within a program such that when the location is reached during program execution, control passes to CID which displays a message and gives control to you.

The SET,BREAKPOINT command (described in section 2) can be used to set breakpoints in the home program or in any other program unit in the user program. For example:

SET,BREAKPOINT,L.100

Sets a breakpoint at line 100 of the home program.

SET,BREAKPOINT,P.ADDSUB_S.12

Sets a breakpoint at the statement labeled 12 in program unit ADDSUB.

It is important to note that breakpoints suspend execution before the statement is executed. For example, assume a program contains the following statements:

```
1    A=0.0
2    A=A+1.0
```

and that a breakpoint is set at line 2. Then when line 2 is reached, execution is immediately suspended, before the statement at line 2 is executed. Thus, A has the value 0.0, not 1.0. When execution is resumed, the statement at line 2 is executed and the value of A is replaced by a new value.

FREQUENCY PARAMETERS

When a breakpoint is set at a statement, execution is suspended each time that statement is reached. For example, if a breakpoint is set at a statement within a DO loop, suspension occurs on each pass through the loop. This can result in many unnecessary suspensions during the course of a debug session. To alleviata this situation, CID provides another form of the SET,BREAKPOINT command that is extremely useful for debugging DO loops and other sections of a program which are executed frequently. The form of this command is:

SET,BREAKPOINT,loc,first,last,step

where first, last, and step are frequency parameters. This command sets a breakpoint that suspends execution every stepth time the breakpoint is reached, beginning with the first time and ending with the last time. For example, the command:

SET,BREAKPOINT,L.50,10,100,5

sets a breakpoint at the statement labeled 50 which is recognized on the tenth time the statement is reached and every fifth time thereafter, up through the hundredth time.

As an example of the use of the frequency parameters, consider the following loop:

```
      DO 8 I=1,1000
      X=X+FX/DX
    8 CONTINUE
```

To examine the progress of the iteration X=X+FX/DX, you can set a breakpoint at statement 8, specifying frequency parameters to suspend execution at an interval rather than on each pass through the loop. For example:

SET,BREAKPOINT,S.8,1,1000,100

sets a breakpoint that suspends execution on every hundredth pass through the loop, starting with the first pass.

To illustrate the SET,BREAKPOINT command, the program shown in figure 3-7 is executed under CID control. This program performs the same calculation as the program shown in figure 2-4 in section 2, but it has been modularized into a main program and a subroutine. The main program RDTR reads input data from the file TRFILE and calls subroutine AREA. AREA performs the computations and returns the final result. Control then branches to the beginning of the program, and another record is read and processed. A sample input file, containing four records, is also shown in figure 3-7.

The resulting debug session is shown in figure 3-8. The purpose of this session is to suspend execution in the main program, immediately before the call to AREA, to examine the input values. Execution is also suspended at the end of subroutine AREA to examine the intermediate values and the final result. To accomplish this, breakpoints are set at line 5 of the main program and at line 7 of subroutine AREA. In both SET,BREAKPOINT commands, the frequency parameters:

1,10,2

are included so that execution is suspended only on every other pass through the program, beginning with the first pass. After the tenth pass, the breakpoint will not be recognized. Since four input records are provided for this example, execution is suspended on the first and third passes through the program. Each time execution is suspended, the PRINT command is used to display the desired values, and the GO command is used to resume execution.

Program RDTR and Subroutine AREA:

```
          PROGRAM RDTR        74/74    OPT=0

                PROGRAM RDTR
                OPEN (UNIT=2,FILE='TRFILE')
                REWIND 2
       10       READ (2,*,END=999) X1,Y1,X2,Y2,X3,Y3
                CALL AREA (X1,Y1,X2,Y2,X3,Y3,A)
                GO TO 10
      999       STOP
                END

       SUBROUTINE AREA        74/74    OPT=0

                SUBROUTINE AREA (X1,Y1,X2,Y2,X3,Y3,A)
                S1=SQRT((X2-X1)**2 + (Y2-Y1)**2)
                S2=SQRT((X3-X1)**2 + (Y3-Y1)**2)
                S3=SQRT((X3-X2)**2 + (Y3-Y2)**2)
                T=(S1+S2+S3)/2.0
                A=SQRT(T*(T-S1)*(T-S2)*(T-S3))
                RETURN
                END
```

Input Data:

```
0.0 0.0 2.0 0.0 0.0 2.0
0.0 1.0 0.5 2.0 -1.0 1.2
6.1 2.0 0.1 -4.0 3.2 7.0
0.2 -2.9 -1.3 8.0 5.6 2.8
```

Figure 3-7. Program RDTR, Subroutine AREA, and Input Data

DISPLAYING A LIST OF BREAKPOINTS

You can display a list of breakpoints defined in a debug session by entering one of the following commands:

LIST,BREAKPOINT

Displays a list of all breakpoints in the program.

LIST,BREAKPOINT,P.prog

Displays a list of all breakpoints in the specified program unit.

The LIST,BREAKPOINT command lists the breakpoints that exist at the time the command is entered. The list contains the number and location of each breakpoint in the following form:

*B #i = P.prog_L.n

where i is the breakpoint number assigned by CID, prog is the program unit containing the breakpoint, and n is the line number where the breakpoint is located. (If the breakpoint was set at a statement label, the notation S.n appears, indicating the statement label, instead of L.n.) If the breakpoint is in the home program, the qualifier P.prog does not appear. If frequency parameters were specified when the breakpoint was set, they also appear in the list.

If no breakpoints exist when a SET,BREAKPOINT command is entered, CID displays the following message:

NO BREAKPOINTS

Figure 3-9 illustrates a debug session (for the program shown in figure 3-7) in which some breakpoints are defined, and listed later in the session.

REMOVING BREAKPOINTS

You can remove breakpoints during a debug session by entering one of the following commands:

CLEAR,BREAKPOINT

Clears all breakpoints in all program units.

CLEAR,BREAKPOINT,loc_1,loc_2,...,loc_n

Clears the breakpoints from the specified locations; loc_i can have any of the following forms:

L.n	Line n of the home program
S.n	Statement labeled n in the home program
P.prog_L.n	Line n in program unit prog
P.prog_S.n	Statement labeled n in program unit prog
P.prog	All breakpoints in program unit prog
#n	Breakpoint having number n

CYBER INTERACTIVE DEBUG	
?set,breakpoint,l.5,1,10,2	Set breakpoint at line 5 of main program. Breakpoint suspends execution on first and third passes.
?set,breakpoint,p.area_l.7,1,10,2	Set breakpoint at line 7 of subroutine AREA. Break-point suspends execution on first and third passes.
?go	Initiate execution.
*B #1, AT L.5	Execution suspended at line 5 of main program.
?print*,x1,y1,x2,y2,x3,y3	Display first set of input values.
0. 0. 2. 0. 0. 2.	
?go	Resume execution.
*B #2, AT P.AREA_L.7	Execution suspended at line 7 of subroutine AREA.
?print*,s1,s2,s3,t,a	Display intermediate values and final result.
2. 2. 2.828427124746 3.414213562373 2.	
?go	Resume execution.
*B #1, AT L.5 (OF P.RDTR)	Execution suspended at line 5 of main program.
?print*,x1,y1,x2,y2,x3,y3	Display third set of input values.
6.1 2. .1 -4. 3.2 7.	
?go	Resume execution.
*B #2, AT P.AREA_L.7	Execution suspended at line 7 of subroutine AREA.
?print*,s1,s2,s3,t,a	Display intermediate values and final result.
8.485281374239 5.780138406647 11.42847321386 12.84694649737 23.7	
?go	Resume execution.
*T #17, END IN P.RDTR_L.6	Program runs to completion.
?	
STOP	
21100 MAXIMUM EXECUTION FL.	
1.080 CP SECONDS EXECUTION TIME.	
quit	Terminate session.
DEBUG TERMINATED	

Figure 3-8. Debug Session Illustrating SET,BREAKPOINT Command

CYBER INTERACTIVE DEBUG	
?set,breakpoint,l.5,1,10,2	Set breakpoint at line 5 of home program.
?set,breakpoint,p.area_l.7,1,10,2	Set breakpoint at line 7 of subroutine AREA.
?go	Initiate execution.
*B #1, AT L.5	Execution suspended at line 5 of home program.
?list,breakpoint	List all breakpoints.
*B #1 = L.5,,10,2, *B #2 = P.AREA_L.7,,10,2	
?	

Figure 3-9. Debug Session Illustrating LIST,BREAKPOINT Command

If the first form of the command is entered, CID displays the message:

*WARN - ALL WILL BE CLEARED
OK?

This message serves as a reminder that the command you have just entered will remove all breakpoints in the entire program (not just the home program). If this is not what you want, enter:

NO

and CID disregards the CLEAR,BREAKPOINT command. If you do want the CLEAR,BREAKPOINT command to be executed, enter:

OK

and CID clears all existing breakpoints.

If a breakpoint does not exist at a specified location, CID displays the message:

NO BREAKPOINT loc

where loc is the braakpoint location, and no action is taken.

Examples:

CLEAR,BREAKPOINT,L.14,L.20,P.SUB3_S.5

Removes the breakpoints from lines 14 and 20 of the home program and from the statement labeled 5 in program unit SUB3.

CLEAR,BREAKPOINT,P.READXY,P.ADDR

Removes all breakpoints from program units READXY and ADDR.

CLEAR,BREAKPOINT,#3,#5,#6

Removes breakpoints 3, 5, and 6.

CLEAR,BREAKPOINT,L.14,P.MULT,#6

Clears the breakpoint from line 14 of the home program, clears all breakpoints from program unit MULT, and clears breakpoint number 6 from the home program.

SUSPENDING EXECUTION WITH TRAPS

A trap is a special condition within a program which suspends program execution whenever that condition is detected during execution. Control then passes to CID, which displays a message and gives control to you.

TRAP USAGE

The most useful traps to the FORTRAN programmer are the LINE and STORE traps. (The END and ABORT traps are also used, but they are established automatically by CID.) The OVERLAY trap is used only with programs containing overlays and is described in section 5. The remaining CID traps (JUMP, FETCH, INS, and RJ) are not recommended for use with FORTRAN programs because their use can be time-consuming and they can cause program execution to suspend in unexpected places. For example, the RJ trap suspends execution at every return jump instruction generated by the FORTRAN program. Return jumps are generated not only by CALL and RETURN statements, but also by any external reference. A program can contain many hidden external references, such as those generated by input/output statements. The result can be many unnecessary suspensions of execution. The traps described in this section are listed in table 3-2. Refer to the CYBER Interactive Debug reference manual for information on other CID traps.

When a trap condition is detected, execution is suspended and CID gains control and issues a message identifying the trap, followed by a ? prompt for user input. The message gives information about the trap, including the trap type, the trap number, and the statement (L.n or S.n) where the trap occurred. The trap number is a decimal integer assigned by CID. Traps are numbered sequentially in the order they are established. The purpose of trap numbers is described later in this section. An example of a trap message is:

*T #3, LINE IN P.SBX_L.5
?

In this example, a LINE trap has been detected in line 5 of program unit SBX; this trap was the third one established by the programmer.

TABLE 3-2. TRAP TYPES

Trap Type	Short Form	Condition	Established By	User Gets Control
LINE	L	Beginning of an executable statement	User	Before the statement is executed
STORE	S	Store to memory	User	After the store
OVERLAY	OVL	Overlay load	User	After the overlay is loaded
INTERRUPT	INT	User interrupt or time limit	Default	After the interrupt
END	E	Normal program termination	Default	After termination
ABORT	A	Abnormal program termination	Default	After termination

In response to the ? prompt, you can enter any CID command. Typically, you will use this opportunity to examine the values of program variables, and make any desired changes to these values. Program execution can be resumed by entering a GO command.

Traps suspend execution when a specific event occurs. Some traps suspend execution before the event, while others suspend execution after the event. This is an important distinction because it can affect the status of variables you are displaying or altering. For example, assume that execution is suspended at line 2 of the following program segment:

```
1        A = 0.0
2        A = A+1.0
```

If the trap suspended execution before the statement at line 2 was executed, A contains 0.0. If the trap suspended execution after the statement was executed, A contains 1.0.

Table 3-2 indicates, for each trap, the point in execution where CID gets control.

The traps described in this section are of two types: user-established and default. The user-established traps are set by the SET,TRAP command. The default traps always exist; it is not necessary to specify a SET,TRAP command for these traps. Table 3-2 indicates default and user-established traps.

DEFAULT TRAPS

CID provides default traps that are automatically set at the beginning of a debug session. These traps allow you to gain control without actually establishing any traps or breakpoints. The default traps are the END, ABORT, and INTERRUPT traps.

Together, the END and ABORT traps transfer control to CID on any program termination. Thus, for the initial debug session, you can allow your program to terminate; by examining the status of the program at the point of termination, you can determine where traps or breakpoints should be set for subsequent sessions.

END Trap

The END trap gives control to CID on normal program termination. This trap always occurs when a program terminates normally, regardless of any CID commands that have been entered to set or clear traps.

Note that the debug session does not end when your program terminates. The END trap allows you to enter commands and continue the session until you enter the QUIT command.

The debug session in figure 2-4 in section 2 illustrates the END trap. The program runs to completion and CID gains control and issues the message:

```
*T #17, END IN L.5
?
```

CID permanently assigns the number 17 to the END trap. In response to the ? prompt, you can display program variables as they exist at the time of termination or you can terminate the session by entering QUIT. You cannot enter a GO command following an END trap.

ABORT Trap

The ABORT trap is useful because it allows you to gain control on any abnormal termination of program execution. The status of program variables can be examined as they exist at the precise time of termination.

To illustrate how the ABORT trap works, a program containing an error is executed under CID control. The source listing and session log are shown in figure 3-10. The statement C=(A+B)/(A-B) results in a division by zero. The variable C is set to an infinite value (represented by the character R) and when C is used as an operand in the next statement, the program aborts with a mode 2 error. CID

```
                    PROGRAM ERR          74/74    OPT=0

             1               PROGRAM ERR
             2               A=2.0
             3               B=2.0
             4               C=(A+B)/(A-B)
             5               D=C+1.0
             6               END


  CYBER INTERACTIVE DEBUG
  ?go <------------------------------------------ Initiate execution.

  *T #18, ABORT CPU ERROR EXIT 02 IN L.5 -------- ABORT trap suspends execution at line 5.
  ?print*,a,b,c,d <------------------------------ Display values of variables.

   2. 2. R R <----------------------------------- Variables C and D contain infinite values.
  ?quit <---------------------------------------- Terminate session.

    DEBUG TERMINATED
```

Figure 3-10. Program ERR and Debug Session Illustrating ABORT Trap

immediately gains control and issues the trap message indicating the trap type, number, location, and the error number. The user enters the PRINT command to display the contents of program variables. Note that, in this case, the value of the infinite operand is represented by the character R and the indefinite operand is represented by the character I. The QUIT command terminates the session.

The ABORT trap is permanently assigned the number 18 by CID.

INTERRUPT Trap

An INTERRUPT trap gives control to CID when a terminal interrupt is issued.

The procedure for issuing a terminal interrupt depends on the terminal type and on the interactive communication system in use. Refer to the Intercom reference manual (NOS/BE) or the IAF reference manual (NOS) for more information on interrupt sequences. A terminal interrupt allows you to gain control at any time during a debug session.

When you enter the appropriate interrupt sequence, the process currently active is interrupted; CID gets control, issues the INTERRUPT trap message, and gives control to you. The INTERRUPT trap can be used to terminate excessive output to the terminal, although it will cause the remaining output to be lost. It can also be used to interrupt a program that you believe to be looping excessively at some unknown location.

USER-ESTABLISHED TRAPS

In addition to the default traps, CID provides traps that can be established whenever you have control.

SET,TRAP Command

The traps described in the following paragraphs are established with the SET,TRAP command. This command has the form:

SET,TRAP,type,scope

where type is one of the trap types listed in table 3-2, and scope is one of the notation forms listed in table 3-1.

The scope parameter of the SET,TRAP command specifies the program locations for which the trap is effective. The scope of a trap can be a single location, such as a variable or a FORTRAN statement, or it can consist of multiple locations, such as an array or a complete program unit. Not all forms listed in table 3-1 are valid for all trap types; valid forms depend on the particular trap type.

Certain traps allow you to specify an asterisk (*) for the scope parameter. This means that the trap is effective throughout the entire program.

Traps can be established whenever program execution is suspended and CID has issued a ? prompt. If a condition for which you have established a trap does not occur, the program executes normally.

LINE Trap

The LINE trap suspends program execution and gives control to CID immediately prior to execution of each executable FORTRAN statement within the specified scope. This trap allows you to step through an executing program, and to examine and alter variable values before each statement is executed. The command to set a LINE trap has the form:

SET,TRAP,LINE,scope

where scope has one of the following forms:

*	Trap is set for each statement in all program units in the user program.
P.prog	Trap is set for each statement in the specified program unit.

Examples:

SET,TRAP,LINE,*

Suspends execution before each executable statement of the entire program.

SET,TRAP,LINE,P.SUBX

Suspends execution before each executable statement in program unit SUBX.

An additional form of the scope parameter is available which allows you to set the LINE trap for a range of FORTRAN lines. The form:

L.n_1...L.n_2

defines the trap for source lines n_1 through n_2. Qualification notation can be used with this form to denote a range of lines in a program unit other than the home program.

Examples:

SET,TRAP,LINE,L.1...L.14

Sets a LINE trap at source lines 1 through 14 of the home program.

SET,TRAP,LINE,P.AAA_L.45...P.AAA_L.54

Sets a LINE trap at source lines 45 through 54 of program unit AAA.

To illustrate the LINE trap, the program in figure 3-11 is executed under CID control. The session log is shown in figure 3-12. The program consists of a main routine PROGY and a subroutine SETB. The main routine contains two calls to SETB; SETB stores values into array B depending on the value of the variable K. The first CID command sets the LINE trap. The scope parameter specifies that the trap applies only to the main program. The trap occurs immediately before each executable statement. The PRINT command is entered after each subroutine call (execution suspended at lines 6 and 8). The GO command resumes execution after each suspension. Note that both the LINE and END traps occur at line 8, the last executable statement of the program. This illustrates that more than one trap can occur at the same location.

```
        PROGRAM PROGY         74/74    OPT=0

  1                       PROGRAM PROGY
  2                       COMMON /BCOM/ B(5)
  3                       N=5
  4                       K=1
  5                       CALL SETB (K,N)
  6                       K=2
  7                       CALL SETB (K,N)
  8                       END

     SUBROUTINE SETB          74/74    OPT=0

  1                       SUBROUTINE SETB (K,N)
  2                       COMMON /BCOM/ B(5)
  3                       IF (K .EQ. 1) THEN
  4                         DO 6 I=1,N
  5               6         B(I)=-1.0
  6                       ELSE
  7                         DO 12 I=1,N
  8               12        B(I)=1.0
  9                       ENDIF
 10                       RETURN
 11                       END
```

Figure 3-11. Subroutine SETB and Main Program

STORE Trap

The STORE trap suspends execution whenever data is stored into the specified locations. The command to set a STORE trap has the form:

SET,TRAP,STORE,scope

where scope has one of the following forms:

var

Simple or subscripted variable var in home program.

P.prog_var

Simple or subscripted variable var in program unit prog.

C.blk

All words in common block blk.

The STORE trap is useful because it allows you to gain control whenever a specific variable is modified. You can then display the value stored into the variable. A variable is modified whenever a statement is executed in which the variable appears to the left of an equals sign or whenever the variable receives data as a result of an input operation.

Examples:

SET,TRAP,STORE,P.SUBX_A

Suspends execution whenever data is stored into the variable A in subroutine SUBX.

SET,TRAP,STORE,ARR(100)

Suspends execution whenever data is stored into word ARR(100) in the home program.

SET,TRAP,STORE,C.BCOM

Suspends execution whenever data is stored into any word in common block BCOM.

If an array name without a subscript is specified for the SCOPE parameter, the STORE trap is set only for the first location of the array. To set a STORE trap that is effective for all elements of an array, or for elements within an array, use the following ellipsis notation:

a(n_1)...a(n_2)

This notation denotes elements n_1 through n_2 of array a.

Qualification notation can be combined with ellipsis notation to designate an array local to program unit other than the home program:

P.prog_a(n_1)...P.prog_a(n_2)

denotes elements n_1 through n_2 of array a in program unit prog.

The following examples assume an array dimensioned X(10):

SET,TRAP,STORE,X

Suspends execution whenever data is stored into X(1).

SET,TRAP,STORE,X...X(10)

Suspends execution whenever data is stored into any of the elements X(1) through X(10).

SET,TRAP,STORE,P.ADDB_X(5)...P.ADDB_X(15)

Suspends execution whenever data is stored into any of the elements X(5) through X(15) in program unit ADDB. (X is local to ADDB.)

The STORE trap can be helpful in debugging situations involving a long program in which a variable is being inadvertently changed at an unknown location. For example, suppose a program contains a variable A which is passed as a parameter to several subprograms. It might be important to determine which subprogram is changing the value of A. The command:

? SET,TRAP,STORE,A

```
CYBER INTERACTIVE DEBUG
?set,trap,line,p.progy
?go
*T #1, LINE AT L.3
?print*,n
 *****************
?go
*T #1, LINE AT L.4
?go
*T #1, LINE AT L.5
?print*,n,k
 5 1
?go
*T #1, LINE AT L.6
?print*,b
 -1. -1. -1. -1. -1.
?go
*T #1, LINE AT L.7
?print*,k
 2
?go
*T #1, LINE AT L.8
?print*,b
 1. 1. 1. 1. 1.
?go
*T #17, END IN L.8
?
      END  PROGY
      12300 MAXIMUM EXECUTION FL.
       .719 CP SECONDS EXECUTION TIME.
```

Set LINE trap in PROGY.

Inititate execution.

Variable N contains meaningless value because it has not been initialized.

LINE trap suspends execution at lines 3 through 8. After each suspension, selected values are displayed and execution is resumed.

END trap occurs at line 8.

Figure 3-12. Debug Session Illustrating LINE Trap

can be used, with the home program set to the highest level routine that uses A. Whenever A is changed, in any of the subprograms, execution will suspend and the trap message will indicate the routine name and line number where the change occurred. Note that this example is valid even if the dummy argument in a subprogram has a name different from A. Thus, the preceding command will cause execution to suspend when the variable B is changed as follows:

```
      .
      .
      .
CALL SUB(A)
      .
      .
      .
END
SUBROUTINE SUB(B)
B=B+1.0
      .
      .
      .
```

In the trap message, however, the variable that appears is the one specified in the SET,TRAP command. One significant disadvantage of the STORE trap is that it requires interpret mode execution. (See Interpret Mode, following this discussion.) After you set a STORE trap, CID displays the message:

INTERPRET MODE TURNED ON

Interpret mode greatly increases the execution time required by a program. For this reason, you should not set a STORE trap until you reach a point in a debug session where you want the trap to be effective. When you reach a point in the session where you no longer need the trap, you should remove it. (See Removing Traps.)

An example of a debug session using a STORE trap is illustrated in figure 3-13. The program in figure 3-11 was executed under CID control to produce this session log. The STORE trap is set so that CID gets control whenever data is stored into common block BCOM. Execution is subsequently suspended on each pass through the DO loop when the constant is stored into each of the five locations of the array B in common block BCOM.

DISPLAYING A LIST OF TRAPS

To display a list of traps defined for a debug session, enter the command:

LIST,TRAP

This command displays the type, number, and location of all traps that exist at the time the command is entered. LIST,TRAP output has the following form:

T #n = type scope

where n is the trap number assigned by CID, type is the trap type as listed in table 3-2, and scope is the location of the trap in the form specified in the SET,TRAP command (P.prog or P.prog_var).

If no traps exist when LIST,TRAP is entered, CID displays the message:

NO TRAPS

Figure 3-14 shows a debug session in which some traps are established and then listed. Trap number 1 is a LINE trap in program unit RDTR, and trap number 2 is a STORE trap for the variable A in program unit AREA.

REMOVING TRAPS

A user-defined trap can be removed at any time during a debug session with the CLEAR,TRAP command. This command has the following forms:

CLEAR,TRAP

Removes all user-defined traps in all program units.

CLEAR,TRAP,type,P.prog

Removes the traps of the specified type from the specified program unit.

CLEAR,TRAP,type

Removes all traps of the specified type.

CLEAR,TRAP,#n_1,#n_2,...,#n_m

Removes the traps identified by the specified numbers.

The type parameter can be any of the types listed in table 3-2 except for the default INTERRUPT, END, and ABORT traps, which cannot be removed.

```
CYBER INTERACTIVE DEBUG
?set,trap,store,c.bcom  <----------  Set STORE trap for common block BCOM.

INTERPRET MODE TURNED ON
?go

*T #1, STORE INTO B IN P.SETB_L.5
?go

*T #1, STORE INTO B+1 IN L.5
?go

*T #1, STORE INTO B+2 IN L.5        STORE trap suspends execution each time a value is stored into
?go                                 array B in common block BCOM. Execution is resumed after
                                    each suspension. Note that B corresponds to the first word of
*T #1, STORE INTO B+3 IN L.5        the block, B+1 corresponds to the second word, and so forth.
?go

*T #1, STORE INTO B+4 IN L.5
?quit

  DEBUG TERMINATED
```

Figure 3-13. Debug Session Illustrating STORE Trap

```
CYBER INTERACTIVE DEBUG
?set,trap,line,p.rdtr  <----------  Set LINE trap in program unit RDTR.

?set,trap,store,p.area_t  <-------  Set STORE trap for variable T in subroutine AREA.

INTERPRET MODE TURNED ON
?go  <----------------------------  Initiate execution.

*T #1, LINE AT L.2
?list,trap  <---------------------  Display trap information.

T #1 = LINE P.RDTR,   T #2 = STORE P.AREA_T
?
```

Figure 3-14. Debug Session Illustrating LIST,TRAP Command

The CLEAR,TRAP command can be used to remove traps that are no longer needed in a debug session. The command is also useful when editing command sequences, as described in section 5. Following are some examples of the CLEAR,TRAP command:

CLEAR,TRAP,STORE,P.SUBX_A

Clears the STORE trap at the variable A in program unit SUBX.

CLEAR,TRAP,LINE

Clears the LINE trap.

CLEAR,TRAP,#2,#4,#5

Clears the traps identified by trap numbers 2, 4, and 5.

A debug session using the CLEAR,TRAP command is illustrated in figure 3-15. The session in figure 3-14 is duplicated except that the CLEAR,TRAP command is issued after the third pass through the loop, allowing the program to run to completion without interruption. Note that the END trap is not removed by the CLEAR,TRAP command.

INTERPRET MODE

The STORE trap requires a mode of execution called interpret mode. In interpret mode, each machine instruction is simulated by CID. Interpret mode is automatically activated when a STORE trap is set. Interpret mode remains on until the trap is cleared by a CLEAR,TRAP command or until explicitly turned off. CID indicates interpret mode by issuing the message:

INTERPRET MODE TURNED ON

Execution in interpret mode is more time-consuming than normal execution. For this reason, you should use STORE traps sparingly. If the debug session requires excessive computer time (a time limit interrupt occurs), you should rerun the session and substitute breakpoints for STORE traps wherever possible.

You can reduce. the amount of execution required for interpret mode by turning interpret mode off while executing portions of a program not currently being debugged. The command to turn off interpret mode is:

SET,INTERPRET,OFF

CID responds with the message:

INTERPRET MODE TURNED OFF

This message also appears when all STORE traps are removed with the CLEAR,TRAP command.

Traps requiring interpret mode become inoperative when interpret mode is turned off. They can be reactivated by the command:

SET,INTERPRET,ON

The use of the SET,INTERPRET command is illustrated by the debug session shown in figure 3-16. The program shown in figure 3-11 is executed in debug mode to produce this session. In this example, a STORE trap, which activates interpret mode, is established for the variable K in the main program. Interpret mode is then turned off while subroutine SETB is executing. To accomplish this, breakpoints are set at the beginning and at the end of the subroutine. When execution is suspended at the first breakpoint, interpret mode is turned off; when execution is suspended at the second breakpoint, interpret mode is turned back on, reactivating the STORE trap.

```
CYBER INTERACTIVE DEBUG
?set,trap,store,c.bcom  <------  Set STORE trap for common block BCOM.

INTERPRET MODE TURNED ON
?go  <------  Inititate execution.

*T #1, STORE INTO B IN P.SETB_L.5
?go
                                          STORE trap suspends execution on
*T #1, STORE INTO B+1 IN L.5      <------  store into array B. Execution resumed
?go                                       after each suspension.

*T #1, STORE INTO B+2 IN L.5
?clear,trap,store  <------  Remove STORE trap.

INTERPRET MODE TURNED OFF
?list,trap

NO TRAPS
?go

*T #17, END IN P.PROGY_L.8  <------  Program runs to completion.
?
       END  PROGY
       12300 MAXIMUM EXECUTION FL.
        .367 CP SECONDS EXECUTION TIME.
```

Figure 3-15. Debug Session Illustrating CLEAR,TRAP Command

```
CYBER INTERACTIVE DEBUG
?set,trap,store,k ◄──────────────── Set STORE trap for variable K.

INTERPRET MODE TURNED ON
?set,breakpoint,p.setb_l.3 ◄─────── Set breakpoint at first executable statement of SETB.

?set,breakpoint,p.setb_l.10 ◄────── Set breakpoint at RETURN statement in SETB.

?go

*T #1, STORE INTO K IN L.4 ◄─────── STORE trap at line 4.
?go

*B #1, AT P.SETB_L.3 ◄───────────── Breakpoint suspends execution at line 3 of SETB.
?set,interpret,off ◄─────────────── Turn off interpret mode. (STORE trap inhibited.)

?go

*B #2, AT P.SETB_L.10 ◄──────────── Breakpoint suspends execution at line 10 of SETB.
?set,interpret,on ◄──────────────── Turn on interpret mode. (STORE trap reactivated.)

?go

*T #1, STORE INTO K IN P.PROGY_L.6 ◄ STORE trap at line 6.
?print*,k

 2
?quit

  DEBUG TERMINATED
```

Figure 3-16. Debug Session Illustrating SET,INTERPRET Command

This method of turning off interpret mode is rather cumbersome since it is necessary to enter the SET,INTERPRET commands on each pass through the subroutine. This can be accomplished more efficiently by including the SET,INTERPRET commands in a command sequence so that they could be executed automatically. Command sequences are described in section 4.

SUMMARY OF TRAP AND BREAKPOINT CHARACTERISTICS

The following is a summary of trap and breakpoint information presented in this section:

- You can set, clear, or list traps and breakpoints any time CID has control and has prompted for user input.
- Only one breakpoint can be established at a single statement; however, a single breakpoint and multiple traps can be set to occur at a single statement.
- Traps and breakpoints exist for the duration of the debug session unless removed by the CLEAR command or inhibited by the SET,INTERPRET,OFF command before the session is terminated.
- The frequency parameters of the SET,BREAKPOINT command can be used to avoid suspending execution on each pass through a loop.
- CID automatically establishes END, ABORT, and INTERRUPT traps so that you receive control on any program termination, even if you have not explicitly established any traps or breakpoints.
- Breakpoints suspend execution before the statement at the breakpoint location is executed. The point in the execution of a statement at which a trap suspends execution depends on the trap type. The statement at the trap or breakpoint location is executed in a normal manner.
- The STORE trap activates INTERPRET mode, which increases execution time. To reduce execution time, specify the SET,INTERPRET,OFF command when executing portions of the program already debugged, or substitute breakpoints for these traps.

DISPLAYING PROGRAM VARIABLES

When execution of your program is suspended and CID has prompted for user input, you can enter commands to display the values of program variables as they exist at the time of suspension. This discussion includes those forms of the display commands that are most useful to the FORTRAN programmer.

CID provides three commands for displaying the values of program variables: the LIST,VALUES command; the PRINT command; and the DISPLAY command. These commands are summarized in table 3-3.

RECOGNIZING ERRONEOUS VALUES

FORTRAN stores specific values into variables to indicate that certain error conditions exist. When these values are displayed with a CID command, the following representations are used (these values are installation-dependent):

TABLE 3-3. DISPLAY COMMANDS

Command	Description	Formatting	Scope
LIST,VALUES	Lists alphabetically all variable names and values within specified scope.	Automatic according to variable type.	Specified program unit; entire program if none specified.
PRINT	Displays contents of specified variables.	Automatic according to variable type.	Home program only.
DISPLAY	Displays contents of specified variable.	User-specified; default is variable type.	Default is home program; variables can be qualified for other than home program.

Value	Description
R or -R	Value out of range: the magnitude of a real value exceeds the limits of the computer.
I	Indefinite: result of an invalid operation.
-I	Negative indefinite: result of an invalid operation, or variable is undefined.
**********	The value exceeds the range of the default format.

When an indefinite or out of range value is used in a computation, the program aborts. If one of these values is displayed as the value of a program variable, it is probably the result of a programming error and should be investigated. Refer to appendix C for an explanation of error conditions and possible causes.

LIST,VALUES COMMAND

The LIST,VALUES command alphabetically lists all variables defined in the source program and the current value of each. This command automatically formats the variables according to the variable type as declared in the source program. The command has the following forms:

LIST,VALUES

Lists all variable names and values defined in all program units, including arrays in their entirety.

LIST,VALUES,P.$name_1$,P.$name_2$,...,P.$name_n$

Lists all variable names and values defined in the specified program units.

The LIST,VALUES command provides a formatted snapshot of the status of program variables; however, it can produce a large amount of output, particularly if the program contains large arrays. In this case, you can send the output to an auxiliary file (described in section 4) or use an alternate command to display the values. Use the PRINT or DISPLAY command to avoid large amounts of output.

LIST,VALUES is also slow and can add substantially to the execution time of a debug session, particularly for programs with many variables. It is well worth the additional time in many cases, but alternative commands should always be considered.

Figure 3-17 shows a program that assigns values to an array of type real and to variables of type character, complex, and boolean. The program calls a subroutine that adds a constant to the array.

```
         PROGRAM TYPES        74/74    OPT=0

  1                    PROGRAM TYPES
  2                    DIMENSION ARR(-5:5)
  3                    CHARACTER STR*6
  4                    COMPLEX C
  5                    BOOLEAN B
  6           C
  7                    DO 8 I=-5,5
  8           8        ARR(I)=REAL(I)
  9                    STR='ABCDEF'
 10                    C=(2,3.4)
 11                    B="HELLO"
 12                    CALL ADDC (ARR, 10.0)
 13                    END

      SUBROUTINE ADDC         74/74    OPT=0

  1                    SUBROUTINE ADDC (X, C)
  2                    DIMENSION X(11)
  3                    DO 100 I=1,11
  4           100      X(I)=X(I)+C
  5                    RETURN
  6                    END
```

Figure 3-17. Program TYPES and Subroutine ADDC

A debug session for this program, illustrating the LIST,VALUES command, is shown in figure 3-18. A breakpoint is set at line 12 so that execution is suspended before the subroutine call. When the breakpoint is encountered, LIST,VALUES is entered. The program values are formatted according to their type. Note that the boolean variable B is displayed in octal format. The variables in subroutine ADDC are undefined because the subroutine has not yet executed. LIST,VALUES is again entered after the program terminates.

PRINT COMMAND

The PRINT command, introduced in section 2, is the most useful of the display commands for the FORTRAN programmer. This command is identical in format and

```
CYBER INTERACTIVE DEBUG
?set,breakpoint,l.12 ◄──────────── Set breakpoint at line 12.

?go ◄──────────── Initiate execution.

*B #1, AT L.12 ◄──────────── Execution suspended at line 12.
?list,values ◄──────────── Display all program variables and their values.

P.TYPES
ARR(-5) = -5.0,   ARR(-4) = -4.0,   ARR(-3) = -3.0,   ARR(-2) = -2.0
ARR(-1) = -1.0,   ARR(0) = 0.0,   ARR(1) = 1.0,   ARR(2) = 2.0
ARR(3) = 3.0,   ARR(4) = 4.0,   ARR(5) = 5.0
B = 10051 41417 55555 55555,   C = (2.0,3.4),   I = 6,   STR = 'ABCDEF'
P.ADDC ◄──────────── Variables in subroutine ADDC are undefined.
C - UNDEFINED,   I - UNDEFINED,   X - UNDEFINED
?go ◄──────────── Resume execution.

*T #17, END IN L.13 ◄──────────── Program terminates.
?
      END  TYPES
      12300 MAXIMUM EXECUTION FL.
        .523 CP SECONDS EXECUTION TIME.
list,values,p.addc ◄──────────── Display variables and values in subroutine ADDC.

P.ADDC
C = 10.0,   I = 12,   X(1) = 5.0,   X(2) = 6.0,   X(3) = 7.0
X(4) = 8.0,   X(5) = 9.0,   X(6) = 10.0,   X(7) = 11.0,   X(8) = 12.0
X(9) = 13.0,   X(10) = 14.0,   X(11) = 15.0
?
```

Figure 3-18. Debug Session Illustrating LIST,VALUES Command

function to the FORTRAN list-directed PRINT statement. The format is:

PRINT*,list

List elements must be separated by commas and can consist of any of the following:

- Simple or subscripted variables
- Array names
- Character strings
- Constants
- FORTRAN expressions not involving exponentiation or functions
- Implied DO loops enclosed in parentheses

Qualification notation cannot be used with the PRINT command. Except for variables declared in common, the PRINT command can only display variables local to the home program. To display variables belonging to another program unit, you must designate a new home program with the SET,HOME command.

To print the contents of an array, you can use the FORTRAN implied DO statement or you can simply specify the array name. For example, if the statement DIMENSION A(10) appears in the source program, then the commands:

PRINT*,A

and

PRINT*,(A(I),I=1,10)

are equivalent. It should be noted, however, that in the case of multidimensioned arrays, specification of the array name causes the elements to be displayed in column order (the order in which they are stored), while the implied DO form can be used to specify a row-order display.

If the implied DO form is used, CID issues a warning message if the index exceeds the dimensioned boundaries of the array. The variable used as the index in the implied DO does not alter a variable of the same name used in the FORTRAN program.

The PRINT command automatically formats each variable according to its type as declared in the source program. To display variables in a format other than that declared in the sourca program, you must use the DISPLAY command.

A debug session illustrating the PRINT command is shown in figure 3-19. This session is identical to the one in figure 3-18, except the PRINT command is used to display variable values. Note that in order to display values defined in subroutine ADDC while execution is suspended in the main program, ADDC must be designated as the home program. This example also illustrates error diagnostics that can occur as a result of an incorrect PRINT command. The first message indicates that an attempt was made to display a variable in a subroutine before the subroutine was executed. The second message indicates that the specified variable is not defined in the home program.

DISPLAY COMMAND

The DISPLAY command displays the contents of specified variables. In most cases, you will be using the PRINT command since it provides for automatic formatting of

```
CYBER INTERACTIVE DEBUG
?set,breakpoint,L.12                          Set breakpoint at line 12.

?go                                           Initiate execution.

*B #1, AT L.12                                Breakpoint defected at lina 12.
?print*,arr,str,c,b                           Display values in main program.

 -5. -4. -3. -2. -1. 0. 1. 2. 3. 4. 5. ABCDEF(2.,3.4)
 10051414175555555555
?print*,str(4:6)//str(1:3)                    Display substring.

 DEFABC
?set,home,p.addc                              Change home program to ADDC.

?print*,x                                     Attempt to display array X fails because subroutina has
                                              not been called.
*ERROR - PARAMETER REFERENCED BEFORE FIRST SUBROUTINE CALL
?go                                           Resume execution.

*T #17, END IN P.TYPES_L.13                   Program runs to completion.
?
      END  TYPES
      12300 MAXIMUM EXECUTION FL.
       .530 CP SECONDS EXECUTION TIME.
print*,x                                      Attempt to display X fails because it is not defined in
                                              the current home program.
*ERROR - NO PROGRAM VARIABLE X
?set,home,p.addc                              Change home program to ADDC.

?print*,x                                     Display X.

 5. 6. 7. 8. 9. 10. 11. 12. 13. 14. 15.
?
```

Figure 3-19. Debug Session Illustrating PRINT Command

variables and is more familiar to FORTRAN programmers. The DISPLAY command, however, offers the following advantages:

- DISPLAY can display values belonging to any program unit, while PRINT can only display values local to the home program.
- DISPLAY allows you to specify the format of each variable, while PRINT performs automatic formatting. In most cases, automatic formatting is more convenient. However, in situations where you want to display the value of a variable in a format other than its declared or implicit format, you must use the DISPLAY command.
- DISPLAY is the only command that can display the values of debug variables (described later in this section).

The DISPLAY command has the following format:

DISPLAY,variable,format

variable — Simple or subscripted variable in the home program, or simple or subscripted variable with program qualifier (P.variable).

format — Optional format indicator; valid values are:

- F Floating-point (real)
- D Double-precision
- I Integer
- C Character
- O Octal

Default is the variable type as declared in the program.

Since the DISPLAY command automatically formats variables, it is necessary to specify the format parameter only when you want to display a variable in a format other than that declared in the FORTRAN program.

Examples:

DISPLAY,Z

Displays the variable Z.

DISPLAY,XARR(3,7),C

Displays word XARR(3,7) in character format.

To display a variable not in the home program, use qualification notation as follows:

```
DISPLAY,P.prog_var,format
```

where prog is the program unit containing variable var. For example:

```
DISPLAY,P.SUBC_A
```

displays variable A in program unit SUBC.

The ability to use qualification notation provides an advantage over the PRINT command which is limited to the home program. The following two examples produce identical results (while execution is suspended at line 12 of routine RA, the value of variable X in routine RB is displayed):

Example 1:

```
  B #1 AT P.RA_L.12
? SET,HOME,P.RB
? PRINT*,X
  5.0
?
```

Example 2:

```
  B #1 AT P.RA_L.12
? DISPLAY,P.RB_X
  X= 5.0
?
```

Unlike the PRINT command, the DISPLAY command displays only the first word when an array name is specified. To display successive words within an array, specify the first and last words separated by three periods (ellipsis notation), as in the following example:

```
DISPLAY,ARR(5)... ARR(10)
```

displays words 5 through 10 of array ARR in default format.

A major disadvantage of the DISPLAY command is that a list of variables cannot be specified. For example, to display the contents of the variables A, B, and C requires the three DISPLAY commands:

```
DISPLAY,A

DISPLAY,B

DISPLAY,C
```

However, this can be accomplished with the single PRINT command:

```
PRINT*,A,B,C
```

Figure 3-20 illustrates a program and debug session in which the DISPLAY command is used to examine variables in their internal representation. The program reads groups of four integers from the terminal and packs each group into a single word by using the SHIFT function and OR operation. Each packed integer occupies a 15-bit field. The program is executed under CID control and allowed to terminate after three sets of integers have been input. When the END trap gives control to CID, the DISPLAY command is used to display the first three words of the array IPACK in octal format showing the four 15-bit fields of each word. The PRINT command cannot be used since each word of IPACK would be automatically formatted as type integer.

ALTERING PROGRAM VALUES

CID provides commands to alter the values of program variables. Although these commands can be used to change the contents of any location within the program field length, as a FORTRAN programmer you will usually be concerned only with changing the contents of variables. After a variable value is changed and execution is resumed, the new value is used.

The commands are:

- Assignment command. This command is identical to the FORTRAN assignment statement. It allows you to evaluate expressions and to insert values into variables in the home program.

- MOVE command. This command can be used to move data from one program unit to another.

ASSIGNMENT COMMAND

The assignment command is identical in form and function to the FORTRAN assignment statement. This command allows you to make corrections to your program as execution proceeds, eliminating the need for recompiling each time an error is discovered. The assignment command has the form:

```
var=expression
```

where var is a simple or subscripted variable, and expression is any valid FORTRAN arithmetic expression not involving functions or exponentiation. The assignment command functions exactly as in FORTRAN: the expression is evaluated and its value is assigned to the variable on the left of the equal sign; the previous contents of the receiving variable are destroyed. You can enter an assignment command whenever CID has prompted for user input. For example, if program execution is suspended and you have detected a variable that has an incorrect or illegal value, you can use the assignment command to assign a new value to the variable. When you resume execution of the program, the new value is used in subsequent computations involving the altered variable.

Expressions used in assignment commands can be any valid FORTRAN expression with the exception of function references and exponentiation. Any valid FORTRAN constant can appear in an expression. The assignment command performs all conversions according to the rules of FORTRAN. An assignment command cannot span more than one line.

The variables used in an assignment command must all be defined in the home program. To reference variables in another program unit, you must specify the SET,HOME command to designate that program unit as the home program. Just as with FORTRAN, variables are local to the program unit in which they are defined and cannot be mixed in an assignment command with variables local to another program unit.

Changes made through the assignment command do not exist beyond the end of the debug session. When a program is reexecuted, either in debug mode or in normal mode, all program variables have the values defined in the original compiled version.

```
          PROGRAM PAK         74/74    OPT=0

                        PROGRAM PAK
                        DIMENSION IPACK(3),I(4)
                        DO 50 N=1,3
                        PRINT*,'INTEGERS? '
                        READ*, (I(J),J=1,4)
                        IPACK(N)=SHIFT(I(4),45).OR.SHIFT(I(3),30).OR.
                       X         SHIFT(I(2),15).OR.I(1)
                 50     CONTINUE
                        END


CYBER INTERACTIVE DEBUG
?go                                                      Initiate execution.

INTEGERS? 4 8 12 1

INTEGERS? 25 6 14 31                                     Input three sets of integers.

INTEGERS? 18 14 7 10

*T #17, END IN L.9                                       Program terminates.
?
      END  PAK
      20300 MAXIMUM EXECUTION FL.
       .104 CP SECONDS EXECUTION TIME.
display,ipack,o,3                                        Display three words of array IPACK in octal format,
                                                         showing packed integers.
IPACK = 00001 00014 00010 00004    00037 00016 00006 00031
    " +2 = 00012 00007 00016 00022
?quit

  DEBUG TERMINATED
```

Figure 3-20. Program PAK and Debug Session Illustrating DISPLAY Command

Following are some examples of assignment commands:

A=B

Replaces the current contents of A by the current contents of B.

M=N+I-1

Evaluates the expression using the current contents of N and I and assigns the value to M.

ARR(I)=X(I)*X(I)+4./3.*(Y+Z)*2.

Evaluates the expression using the current values of X, I, Y, and Z and assigns the value to the I^{th} word of ARR.

STR=CHARA(1:3)//CHARB(4:6)

Evaluates the character expression and assigns the result to STR. STR, CHARA, and CHARB must be type character.

Figure 3-21 shows a program and debug session illustrating the assignment command. The program calculates the mean of 10 numbers. The program contains a bug: the statement AV=SUM*10.0 should be AV=SUM/10.0.

To enable the program to execute correctly, a breakpoint is set at the PRINT statement. When execution is suspended at this location, the program has already calculated an incorrect value for AV. The assignment command is then used to calculate the correct value of AV. The new value is used in the subsequent PRINT statement when execution is resumed. The erroneous statement must be replaced by the programmer in the corrected version of the source program.

Some additional examples of assignment commands are illustrated in the examples at the end of this section.

MOVE COMMAND

The MOVE command moves data from one program location to another. In most cases, you should use the assignment command to replace the value of a variable by the value of another variable. However, the assignment command can only move one data item at a time, and the move is limited to the home program. The MOVE command allows you to move large blocks of data, such as in arrays and common blocks. In addition, the MOVE command can transfer data between program units. The MOVE command has the form:

MOVE,source,destination,n

```
          PROGRAM AVG         74/74    OPT=0

        1                 PROGRAM AVG
        2                 DIMENSION X(10)
        3                 DATA X/1.0,15.3,2.4,12.7,6.0,
        4                *        5.5,10.1,9.4,4.8,2.0/
        5                 SUM=0.0
        6                 DO 12 I=1,10
        7                 SUM=SUM+X(I)
        8           12    CONTINUE
        9                 AV=SUM*10.0
       10                 PRINT 100, (X(I),I=1,10),AV
       11          100    FORMAT (' NUMBERS: ',10F5.2,/' MEAN: ',F5.2)
       12                 END



CYBER INTERACTIVE DEBUG
?set,breakpoint,L.10                     Set breakpoint to suspend execution at PRINT statement.

?go

*B #1, AT L.10                           Execution suspended.
?print*,av                               Display value of AV.

 692.
?av=sum/10.0                             Calculate correct value for AV.

?print*,av                               Display new value of AV.

 6.92
?go                                      Resume execution.

NUMBERS:  1.0015.30 2.4012.70 6.00 5.5010.10 9.40 4.80 2.00
MEAN:  6.92                              Program prints new value of AV.
*T #17, END IN L.12
?
      END  AVG
      16700 MAXIMUM EXECUTION FL.
       .264 CP SECONDS EXECUTION TIME.
quit

  DEBUG TERMINATED
```

Figure 3-21. Program AVG and Debug Session Illustrating Assignment Command

where source and destination are array or common block specifications (as described in table 3-1) and n is an optional item count. This command moves n items from successive locations starting with the first word of source to successive locations starting with the first word of destination. If n is omitted, one data item is moved. The item count n should not exceed the dimensioned size of either the source or the destination arrays.

To move data between arrays within the home program, specify the array names for the source and destination options. For example:

MOVE,A,B,10

moves the contents of the first ten words of array A to the corresponding locations in array B; both A and B must be defined in the home program.

To specify a starting location other than the first word of the source or destination arrays, use the familiar subscripting notation of FORTRAN. For example:

MOVE,ALPHA(10),BETA(2),100

moves 100 data items from successive locations starting with word 11 of array ALPHA to successive locations starting with word 2 of array BETA.

To move data to or from an array in a program unit other than the home program, use the qualification notation:

P.prog_arr

where prog is the program unit name, and arr is the array name. For example:

MOVE,P.SUBX_BUF,P.SUBY_DAT(100),50

moves 50 data items from successive locations starting with the first word of array BUF in program unit SUBX to successive locations starting with the hundredth word of array DAT in program unit SUBY.

When moving data to or from a common block, you must specify the starting location of the block by using the notation:

C.blk_n

where blk is the common block name and n is the (n+1)th word of the block. Thus, the first word of the common block is indicated by a value of zero for n, the second word is indicated by a value of 1, and so forth.

Examples:

MOVE,BCOM_0,P.FOX_AR,3

> Moves 3 data items from successive locations starting with the first word of common block BCOM to successive locations starting with the first word of array AR in program unit FOX.

MOVE,C.BCOM_1,C.ZCOM_9,50

> Moves 50 data items from successive locations starting with the second word of common block BCOM to successive locations starting with the tenth word of common block ZCOM.

Data can also be moved to or from blank common (C.) or extended memory (XC.blk). Note that a common block specification is independent of the home program; that is, a common block specified in a MOVE command need not be defined in the home program.

Figure 3-22 shows a sample program and debug session illustrating the MOVE command. The program consists of a main program MOVDAT and a subroutine SUB1. MOVDAT defines three arrays and a common block, initializes the common block, and calls SUB1. SUB1 adds the values in array X to the values in array Y, and stores

```
              PROGRAM MOVDAT      74/74    OPT=0

        1                    PROGRAM MOVDAT
        2                    COMMON /BLKA/ A,B,C(5),D(3),E(10)
        3                    DIMENSION R(10),S(10),T(10)
        4                    DATA A,B,C,D,E/3.14,0.16,5*0.0,2.41,8.36,
        5                   *     4.0,10*2.0/
        6              C
        7                    N=10
        8                    CALL SUB1 (N,R,S,T)
        9                    END

        1                    SUBROUTINE SUB1 (N,X,Y,Z)
        2                    DIMENSION X(N),Y(N),Z(N)
        3                    DO 100 I=1,N
        4              100   Z(I)=X(I)+Y(I)
        5                    RETURN
        6                    END

 CYBER INTERACTIVE DEBUG
 ?set,breakpoint,p.sub1_l.3           Set breakpoint at line 3 of subroutine SUB1.

 ?go                                  Initiate execution.

 *B #1, AT P.SUB1_L.3                 Execution suspended at line 3 of SUB1.
 ?print*,'x= ',x,' y= ',y             Display values of arrays X and Y; values are meaningless because X and Y
                                      have not been initialized.
  X= -I -I -I -I -I -I -I -I -I -I  Y= -I -I -I -I -I -I -I -I -I -I
 ?move,c.blka_0,x,10                  Move 10 words from BLKA to X, starting at word 1 of BLKA.

 ?move,c.blka_10,y,10                 Move 10 words from BLKA to Y, starting at word 11 of BLKA.

 ?print*,'x= ',x,' y= ',y             Display values of X and Y.

  X= 3.14  .16 0. 0. 0. 0. 0. 2.41 8.36 4.  Y= 2. 2. 2. 2. 2. 2. 2. 2. 2.

  2.
 ?go                                  Resume execution.

 *T #17, END IN P.MOVDAT_L.9          Program runs to completion.
 ?
       END  MOVDAT
       12400 MAXIMUM EXECUTION FL.
        .643 CP SECONDS EXECUTION TIME.
```

Figure 3-22. Program MOVDAT and Debug Session Illustrating MOVE Command

the results in array Z. If the program were allowed to run to completion, it would produce meaningless results because the arrays R and S, input to SUB1, are not initialized. In the debug session, the MOVE command is used to move data from the common block to the uninitialized arrays so that the computation will yield valid results.

Initially, a breakpoint is set at line 3 of SUB1 so that execution is suspended before the addition is performed. When execution is suspended, the PRINT command shows that arrays X and Y contain meaningless values. The MOVE command is then used to move the first ten values in common to X, and the next ten values to Y. The PRINT command shows that X and Y now contain valid values.

DISPLAYING CID AND PROGRAM STATUS INFORMATION

The following paragraphs describe some CID features and commands that allow you to obtain various kinds of information about the current debug session. These features include:

- Debug variables that contain useful information about the current session; the values of these variables can be displayed at the terminal

- LIST commands that can display such things as load map information, and trap and breakpoint information

DEBUG VARIABLES

CID provides variables that contain information about the current status of a debug session and of the executing program. You can display the contents of debug variables whenever you have control. CID updates these variables, and you cannot alter their contents directly.

Although the debug variables are primarily intended for use by assembly language programmers, some of them can provide information useful to FORTRAN programmers. Those variables that are most useful to FORTRAN programmers are listed in table 3-4. Refer to the CID reference manual for a description of other debug variables.

TABLE 3-4. DEBUG VARIABLES

Variable	Description
#LINE	Number of FORTRAN line executing at time of suspension.
#PC	Previous contents; on STORE trap, #PC contains the value previously stored in the trapped variable.
#HOME	Home program name (P.name).
#BP	Number of existing breakpoints.
#TP	Number of existing traps.
#GP	Number of existing groups.

The #LINE variable contains the number of the FORTRAN source line that was executing at the time of suspension. The form of the line number is P.name L.n, where the Underscore (_) indicates a relative address in a program module or common block. CID normally prints this for you automatically when a trap or breakpoint occurs, but you might wish to display the value yourself at times, especially when using command sequences.

The #HOME variable contains the name of the current home program. The form of this name is P.name. This variable is useful for programs that contain multiple program units. Note that the program name displayed by the #HOME variable might be different than the program name displayed by the #LINE variable since the home program can be changed by the SET,HOME command.

The #PC variable can be displayed after execution has been suspended by a STORE trap. #PC contains the previous value of the variable and can be displayed only when CID is in interpret mode. However, since the STORE trap automatically activates interpret mode, it is not necessary to enter a SET,INTERPRET command before displaying this variable.

The #BP, #TP, and #GP variables contain the numbers of breakpoints, traps, and groups, respectively, currently defined for the debug session. These variables are especially useful for longer, more complex debug sessions.

To display the contents of a debug variable, you must use the DISPLAY command; debug variables cannot be displayed with the PRINT command or LIST,VALUES command. All variables except #PC are automatically displayed in the appropriate format. Since #PC contains a numeric value, you should specify the desired format on the DISPLAY command. Octal format is the default.

Example:

DISPLAY,#LINE

Displays the current source line number in the form P.name L.n.

A debug session using debug variables is ilustrated in figure 3-23. The program executed to produce this session is shown in figure 3-7. In this example, two breakpoints and a STORE trap are defined. While execution is suspended, the DISPLAY command is used to display the values of various debug variables. Note that when #PC is displayed, the F option is specified so that the values are displayed in decimal format.

LIST COMMANDS

The LIST commands allow you to list various types of information relevant to the current debug session or to your program. The LIST commands are summarized in table 3-5.

The LIST commands are particularly useful with longer debug sessions in which you are constantly changing the status of the session. For example, you can initially set some traps or breakpoints, clear some or all of them later in the session, and set new ones; or you can change output options several times during the course of a session. With the LIST commands you can keep track of this and other CID information. The LIST,BREAKPOINT and LIST,TRAP commands are described earlier in this section; the LIST,GROUP command is described in section 4.

```
CYBER INTERACTIVE DEBUG
?set,breakpoint,l.5

?set,breakpoint,p.area_l.2

?go

*B #1, AT L.5
?display,#home  <------------------  Display name of current home program.

#HOME = P.RDTR
?go

*B #2, AT P.AREA_L.2
?set,trap,store,a

INTERPRET MODE TURNED ON
?display,#bp  <--------------------  Display current number of breakpoints.

#BP = 2
?display,#tp  <--------------------  Display current number of traps.

#TP = 1
?display,#home  <------------------  Display name of current home program.

#HOME = P.AREA
?go

*T #1, STORE INTO A (OF P.RDTR) IN L.6
?display,#line  <------------------  Display number of line where execution is suspended.

#LINE = P.AREA_L.6
?display,#pc,f  <------------------  Display previous contents of changed variable in floating
                                     point format.
#PC = -I  <------------------------  Variable A contained meaningless value.
?clear,breakpoint,#2

?go

*T #1, STORE INTO A (OF P.RDTR) IN L.6
?display,#pc,f  <------------------  Display previous contents of changed variable in floating
                                     point format.
#PC = 2.4142135623731
?
```

Figure 3-23. Debug Session Illustrating Debug Variables

TABLE 3-5. LIST COMMANDS

Command	Description
LIST,BREAKPOINT	Lists breakpoint information.
LIST,TRAP	Lists trap information.
LIST,GROUP	Lists command group information.
LIST,MAP	Lists load map information.
LIST,STATUS	Lists information about current status of debug session.
LIST,VALUES	Lists names and values of user-defined variables.

Some of the LIST commands can produce a large volume of output. It is possible to prevent this output from appearing at the terminal and to write it, instead, to a separate file that can then be printed. The commands to accomplish this are described later in this section under Control of CID Output.

LIST,MAP Command

The LIST,MAP command displays load map information. This command is useful when the FORTRAN program contains many subroutine calls or common blocks, since it provides a concise list of subroutine and common block names. The LIST,MAP command can also provide length information which is useful in detecting incorrectly-specified common block lengths. This command has the following forms:

LIST,MAP

> Lists all modules (program units and common blocks) in the user field length. The list includes FORTRAN library modules as well as user-defined modules.

LIST,MAP,P.name$_1$,P.name$_2$,...,P.name$_n$

> Lists the first word address (FWA), length (octal words), and all entry point names for the specified program units.

LIST,MAP,C.name$_1$,C.name$_2$,...,C.name$_n$

> Lists the first word address and length (octal words) of the specified common blocks.

Common blocks are enclosed in slashes in LIST,MAP output.

Figure 3-24 illustrates a debug session for a program in which two common blocks are declared. An incorrect dimension is specified for common block ACOM in subroutine BAKER. The LIST,MAP command displays the correct length as declared in the main program.

LIST,STATUS Command

The LIST,STATUS command displays a brief summary of the status of a debug session as it exists at the time the command is issued. This command has the form:

LIST,STATUS

Information displayed by the LIST,STATUS command includes:

- Home program name.
- Number of breakpoints currently defined.
- Number of traps currently defined.
- Number of groups currently defined.
- Veto mode on or off.
- Interpret mode on or off.
- Output options. Output options are controlled by the SET,OUTPUT command, described under Control of CID Output, which specifies the types of CID output sent to the terminal.
- Auxiliary file options. These options are specified by the SET,AUXILIARY command (described later in this section), which defines an auxiliary output file and specifies the type of output to be sent to that file.

Figure 3-25 illustrates LIST,STATUS commands entered during a debug session. The command is issued at the beginning of the session and again when execution is suspended at a breakpoint. The output indicates the changes in the status of the debug session.

CONTROL OF CID OUTPUT

The output produced by commands such as LIST, TRACEBACK, DISPLAY, and PRINT can become voluminous. As an alternative to displaying all CID output at the terminal, CID provides commands that enable you to prevent specific types of CID output from being displayed

```
      PROGRAM ABLE
      COMMON /ACOM/ A(10),AA(10)
      COMMON /BCOM/ B(50),BB(100)
      CALL BAKER
      END
C
      SUBROUTINE BAKER
      COMMON /ACOM/ X(25)
      COMMON /BCOM/ Y(1)
      DO 6 I=1,25
    6 X(I)=0.0
      DO 8 I=1,150
    8 Y(I)=0.0
      RETURN
      END
```

```
 CYBER INTERACTIVE DEBUG
? list,map                  user programs
 DBUG.,   ABLE,   /ACOM/,   /BCOM/,   BAKER,   FORSYS=,   /Q5.IO./
 /AP.IO./,   /FCL.C./,   /FCL=ENT/,   Q5RPV=,   Q5NTRY=,   /STP.END/
 CHMOVE=,   FCL=FDL,   FORUTL=,   GETFIT=,   SYSAID=,   CPUCPM,   CPU.SYS
 CMF.ALF,   CMF.CSF,   CMM.FFA,   CMF.FRF,   CMF.GSS,   CMM.MEM,   CMM.R
 CMF.SLF,   FDL.RES,   /FDL.COM/,   FDL.MMI,   UCLOAD,   CTL$RM,   CTL$WR
 ERR$RM,   LIST$RM,   RM$SYS=
? list,map,c.acom <-------------------- Display starting address and length of ACOM.
 BLOCK - ACOM,   FWA = 3242B,   LENGTH = 24B <-- Correct length is 24B, or 20 (decimal).
? list,map,c.bcom <-------------------- Display starting address and length of BCOM.
 BLOCK - BCOM,   FWA = 3266B,   LENGTH = 226B
?
```

Figure 3-24. Program ABLE and Debug Session Illustrating LIST,MAP Command

at the terminal, and to define an output file and specify the types of CID output to be written to the file. These commands are as follows:

SET,OUTPUT

Specifies the types of output to be displayed at the terminal.

SET,AUXILIARY

Defines an auxiliary output file and specifies the types of output to be sent to the file.

TYPES OF OUTPUT

For purposes of the SET,OUTPUT and SET,AUXILIARY commands, CID output is classified as to type, with each type represented by a one-letter code. The output codes, along with a description of each code, are listed in table 3-6.

TABLE 3-6. CID OUTPUT TYPES

Output Code	Description
E	Error messages.
W	Warning messages.
D	Output produced by execution of CID commands. Includes output produced by LIST, DISPLAY, PRINT, and TRACEBACK commands.
I	Informative messages. Includes trap and breakpoint messages.
R	Group and file command sequences; output when a READ command is executed.
B	Trap and breakpoint body command sequences; output when a trap or breakpoint with a body is encountered.
T	Echo of user-input information.

SET,OUTPUT COMMAND

The SET,OUTPUT command specifies the types of output to be displayed at the terminal. The SET,OUTPUT command has the form:

SET,OUTPUT,t_1,t_2,...,t_7

where t_i is one of the output codes listed in table 3-6.

Including an output code in the option list of the SET,OUTPUT command causes the associated output type to be displayed at the terminal. Omitting an output code from the option list suppresses the associated output type. Thus, when a SET,OUTPUT command is specified, any output type not included in the option list is not displayed at the terminal. For example, the command:

SET,OUTPUT,E,W,I

causes output types E, W, and I to be displayed at the terminal while it suppresses types D, R, and B.

When the list is omitted, the default options are E, W, D, and I for Interactive Jobs, and E, W, D, I, R, B, and T for Batch Jobs. If a SET,OUTPUT command is not entered, these output types are displayed at the terminal. It is unnecessary to specify type T in a SET,OUTPUT command since all user input is displayed at the terminal when it is entered.

The only output types not automatically displayed are group and file command sequences (type R) and trap and breakpoint bodies (type B). To display this output, in addition to the default types, enter the command:

SET,OUTPUT,E,W,I,D,R,B

If you specify the R option on the SET,OUTPUT command, whenever a READ command is executed, the command sequence is displayed at the terminal. If you specify the B option, whenever a trap or breakpoint for which you have defined a body is detected, the commands comprising the body are displayed. Command sequences are discussed in section 4.

The only output types that cannot be suppressed are the informative messages issued when traps or breakpoints are detected (these are included in type I). These messages are always displayed, regardless of SET,OUTPUT specifications. Error messages (type E) can be suppressed only if you have provided for writing them to an auxiliary file with the SET,AUXILIARY command. If you attempt to suppress error messages and you have not provided for writing them to an auxiliary file, CID issues an error message.

```
 CYBER INTERACTIVE DEBUG
? list,status
 HOME = P.MAIN,    NO BREAKPOINTS,    NO TRAPS,    NO GROUPS,    VETO OFF
 INTERPRET OFF,    OUT OPTIONS = I W E D,    AUXILIARY CLEAR
          •
          •
          •
? list,status
 HOME = P.NEWT,    2 BREAKPOINTS,    1 TRAPS,    NO GROUPS,    VETO OFF
 INTERPRET ON,    OUT OPTIONS = I W E D,    AUXILIARY CLEAR
?
```

Figure 3-25. Debug Session Illustrating LIST,STATUS Command

If you suppress warning messages by omitting W from the SET,OUTPUT command, CID executes all commands that would normally generate a warning message. No user prompt is issued; CID takes the corrective action described in the warning message, responding as if you had entered a YES or OK response (described earlier in this section under Error and Warning Processing).

To suppress all output to the terminal (except trap and breakpoint messages), you can issue either a SET,OUTPUT command with no option list or the command:

```
CLEAR,OUTPUT
```

Prior to entering either of these commands, however, you must provide for writing error messages to an auxiliary file.

After a CLEAR,OUTPUT command has been issued, you can restore output to default conditions with the command:

```
SET,OUTPUT,E,W,D,I
```

The SET,OUTPUT command can be used in conjunction with the SET,AUXILIARY command to suppress certain types of output to the terminal and to send that output type to an auxiliary file. The most common output to suppress is type D, output produced by execution of CID commands. This includes output produced by the LIST and display commands, all of which can produce large amounts of output.

SET,AUXILIARY COMMAND

The SET,AUXILIARY command defines an auxiliary output file and specifies which types of CID output are to be written to that file. The SET,AUXILIARY command has the following form:

SET,AUXILIARY,lfn,t_1,t_2,...,t_7

where lfn is the name of the auxiliary file and t_n is one of the output codes listed in table 3-6.

The SET,AUXILIARY command has no effect on output that is being displayed at the terminal. For example, the command:

```
SET,AUXILIARY,FAUX,I,D
```

creates a file named FAUX and writes all informative and command output messages to the file. These messages are also displayed at the terminal unless the appropriate SET,OUTPUT command has been used to suppress these output types.

The option specifications for an auxiliary file can be changed simply by entering another SET,AUXILIARY command that specifies file name and a new option list; it is not necessary to close the file beforehand.

Only one auxiliary file can be in use at a time. The QUIT command closes the auxiliary file currently in use. To close an auxiliary file before the end of a debug session, enter the command:

```
CLEAR,AUXILIARY
```

An auxiliary file can be closed at any time during a debug session.

After you close an auxiliary file, you can dispose of it in any manner you wish by displaying it at the terminal, sending it to a printer, or storing it on a permanent storage device. CLEAR,AUXILIARY does not rewind the file; after issuing a CLEAR,AUXILIARY you can issue a SET,AUXILIARY for the same file in the same or in a subsequent session, and the additional information is written after the end-of-record.

A common use of the SET,AUXILIARY command is to preserve a copy of a debug session log. For example, the command:

```
SET,AUXILIARY,OUTF,E,W,D,I,T
```

issued at the beginning of a debug session, writes the output types E, W, D, I, and T to file OUTF, thus creating a copy of the session exactly as displayed at the terminal. Note that user commands are automatically echoed at the terminal. However, when outputting to an auxiliary file, you must specify the T option to include user-entered commands in the file.

The following example illustrates a SET,OUTPUT command used in conjunction with a SET,AUXILIARY command to suppress output to the terminal and write it to an auxiliary file:

```
?SET,OUTPUT,E,W,I
?SET,AUXILIARY,LGF,D
?LIST,MAP
?CLEAR,AUXILIARY
?SET,OUTPUT,E,W,I,D
```

This example suppresses all output produced by CID commands (type D), creates an auxiliary file called LGF to which this output is to be written, writes load map information to LGF, closes LGF, and resets output options to original conditions.

The following example illustrates a CLEAR,OUTPUT command used with a SET,AUXILIARY command:

```
?SET,AUXILIARY,AUXF,D,E
?CLEAR,OUTPUT
?LIST,VALUES
?CLEAR,AUXILIARY
?SET,OUTPUT,E,W,D,I
```

This example defines an auxiliary file named AUXF to receive error messages and output from CID commands, turns off output to the terminal (except for trap and breakpoint messages), writes program variables and contents to AUXF, closes AUXF, and restores terminal output to normal default conditions.

An example of a debug session using an auxiliary file is illustrated in figure 3-26. This session was produced by executing subroutine AREA (figure 3-4) under CID control. In this example, an auxiliary file AFILE is defined; the D option causes output from CID commands to be sent to AFILE. A breakpoint is set at line 7 of subroutine AREA, and output to the terminal is suppressed. (Note, however, that the breakpoint message still appears.) On each pass through subroutine AREA, the breakpoint suspends execution, and LIST,VALUES is entered to write all variable names and values to the auxiliary file. After the third pass through AREA, normal output conditions are restored, the value of the variable A is displayed, and the session is terminated. File AFILE (figure 3-27) contains the output from the LIST,VALUES command. (A better way of doing this would be to include the SET,OUTPUT and LIST,VALUES commands in a breakpoint body. This would preclude the necessity of reentering these commands on each pass through the subroutine. Breakpoint bodies are described in section 4.)

```
CYBER INTERACTIVE DEBUG
?set,auxiliary,afile,d,e        Establish auxiliary file AFILE and send all command output and error
                                messages to this file.
?set,breakpoint,p.area_l.7

?clear,output                   Suppress output to terminal.

?go

*B #1, AT P.AREA_L.7
?list,values,p.area

?go

*B #1, AT P.AREA_L.7            List variables and values while execution is suspend on each pass through
?list,values,p.area             subroutine AREA. Output is written to AFILE.

?go

*B #1, AT P.AREA_L.7
?list,values,p.area

?go

*B #1, AT P.AREA_L.7
?set,output,e,w,d,i             Restore normal output to terminal.

?print*,a                       Display value of A. This value is also written to AFILE.

 33.705
?quit

  DEBUG TERMINATED
```

Figure 3-26. Debug Session Illustrating SET,AUXILIARY, SET,OUTPUT, and CLEAR,OUTPUT Commands

```
1                                                 CYBER INTERACTIVE DEBUG
0
 *B #1, AT P.AREA_L.7
 P.AREA
 A = 2.0,    S1 = 2.0,    S2 = 2.0,    S3 = 2.8284271247462                     First pass
 T = 3.4142135623731,   X1 = 0.0,   X2 = 2.0,   X3 = 0.0,   Y1 = 0.0
 Y2 = 0.0,   Y3 = 2.0
 *B #1, AT P.AREA_L.7
 P.AREA
 A = .75,    S1 = 1.1180339887499,    S2 = 1.4142135623731,    S3 = 1.5        Second pass
 T = 2.0161237755615,   X1 = 0.0,   X2 = .5,   X3 = -1.0,   Y1 = 1.0
 Y2 = 2.0,   Y3 = 2.0
 *B #1, AT P.AREA_L.7
 P.AREA
 A = 23.700000000001,   S1 = 8.4852813742386,   S2 = 5.7801384066474
 S3 = 11.428473213864,   T = 12.846946497375,   X1 = 6.1,   X2 = .1           Third pass
 X3 = 3.2,   Y1 = 2.0,   Y2 = -4.0,   Y3 = 7.0
 *B #1, AT P.AREA_L.7
  33.705
```

Figure 3-27. Listing of Auxiliary File AFILE

INTERACTIVE INPUT

Programs receiving input from the terminal can be executed under CID control. A program that is to receive input from the terminal should be written in such a way as to differentiate between a program request for input and a CID request for input. Likewise, you should have some method of distinguishing program output from CID output. This is particularly important when you are running programs under NOS since the system automatically inserts a ? prompt (identical to the CID prompt) at the beginning of a line to indicate a program request for user input.

Examples of debug sessions for programs that receive input from the terminal are illustrated in figures 3-28 (NOS) and 3-29 (NOS/BE). Program ATR reads the coordinates of the vertices of a triangle and calculates the area of the triangle. Files INPUT and OUTPUT are used so that input and output can be performed through the terminal. Immediately before the READ is executed, a WRITE statement displays two asterisks (**) to indicate that the program is waiting for user input. Input data is then entered on the same line as the asterisks. After the final calculation, a WRITE statement displays a message and the calculated area.

The NOS session is more complicated because the system-issued ? prompt is identical to the CID prompt. The two successive asterisks, however, identify the subsequent ? character as being issued by NOS and not by CID.

DEBUGGING EXAMPLES

The following paragraphs present some examples of interactive debugging using the commands described in this section.

SAMPLE PROGRAM CORR

The program entitled CORR reads pairs of numbers and calculates the corralation coefficient of the numbers. The source listing is shown in figure 3-30.

The correlation coefficient is a means of measuring the degree of statistical correlation between two sets of numbers. The formula for the correlation coefficient is:

$$r = \frac{n\Sigma xy - \Sigma x \Sigma y}{\sqrt{n\Sigma x^2 - (\Sigma x)^2}\ \sqrt{n\Sigma y^2 - (\Sigma y)^2}}$$

r	Correlation coefficient.
n	Number of pairs to be correlated.
x,y	Values to be correlated.

The correlation coefficient can have any value between -1 and 1. A coefficient with a magnitude close to 1 indicates close correlation.

The program in figure 3-30 contains a number of bugs. The program compiles successfully, but does not run to completion.

```
00100 PROGRAM ATR
00110 10 PRINT*, ' ** '
00120 READ (*,*,END=999) X1,Y1,X2,Y2,X3,Y3
00130 S1=SQRT((X2-X1)**2 + (Y2-Y1)**2)
00140 S2=SQRT((X3-X1)**2 + (Y3-Y1)**2)
00150 S3=SQRT((X3-X2)**2 + (Y3-Y2)**2)
00160 T=(S1+S2+S3)/2.0
00170 A=SQRT(T*(T-S1)*(T-S2)*(T-S3))
00180 PRINT*, ' AREA IS ',A
00190 GO TO 10
00200 999 STOP
00210 END
```

```
 CYBER INTERACTIVE DEBUG
? set,breakpoint,L.170                 Set breakpoint at line 170.
? go                                   Initiate execution.
  **                                   Program writes asterisks.
? 1.0 2.4 -5.1 0.4 -2.2 0.9            System issues prompt.  User enters input data.
 *B #1, AT L.170                       Execution suspended at line 170.
? print*,s1,s2,s3
  6.419501538282 3.534119409414 2.942787793912
? go                                   Resume execution.
  AREA IS 1.375                        Program writes output.
  **
?                                      User enters carriage return to indicate end-of-input.
 *T #17, END IN L.200                  Program terminates.
? quit

SRU       7.175 UNTS.

 RUN COMPLETE.
```

Figure 3-28. Program ATR and Debug Session Illustrating Interactive Input Under NOS

```
 1                PROGRAM ATR
 2          10    PRINT*, ' ** '
 3                READ (*,*,END=999) X1,Y1,X2,Y2,X3,Y3
 4                S1=SQRT((X2-X1)**2 + (Y2-Y1)**2)
 5                S2=SQRT((X3-X1)**2 + (Y3-Y1)**2)
 6                S3=SQRT((X3-X2)**2 + (Y3-Y2)**2)
 7                T=(S1+S2+S3)/2.0
 8                A= SQRT(T*(T-S1)*(T-S2)*(T-S3))
 9                PRINT*, ' AREA IS ',A
10                GO TO 10
11          999   STOP
12                END

CYBER INTERACTIVE DEBUG
?set,breakpoint,l.8                      Set breakpoint at line 8.
?go                                      Initiate execution.
                                         Program writes asterisks.
 **1.0 2.4 -5.1 0.4 -2.2 0.9             User enters input data.

*B #1, AT L.8                            Execution suspended at line 8.
?print*,s1,s2,s3
 6.419501538282 3.534119409414 2.942787793912
?go                                      Resume execution.

 AREA IS 1.375                           Program writes output.
 **%eof                                  User enters end-of-input indicator.

*T #17, END IN L.3                       Program terminates.
?
```

Figure 3-29. Program ATR and Debug Session Illustrating Interactive Input Under NOS/BE

```
            PROGRAM CORR
      C   CORR CALCULATES A CORRELATION COEFFICIENT
            DIMENSION X(5),Y(5)
      C
      C...INITIALIZATION
      C
            N=1
            SUMX=0.0
            SUMY=0.0
            SUMXSQ=0.0
            SUMXY=0.0
      C
      C...READ NUMBERS TO BE CORRELATED
      C
            OPEN (UNIT=2,FILE='CORFIL')
      10    READ (2,*,END=20) X(N),Y(N)
            N=N+1
            GO TO 10
      C
      C...CALCULATE CORRELATION COEFFICIENT
      C
      20    IF (N.EQ.0) THEN
              PRINT*,' EMPTY INPUT FILE'
            ELSE
              DO 30 I=1,N
              SUMX=SUMX+X(I)
              SUMY=SUMY+Y(I)
              SUMXSQ=SUMXSQ+X(I)**2
              SUMYSQ=SUMYSQ+Y(I)**2
              SUMXY=X(I)+Y(I)
      30      CONTINUE
              NUM=(N*SUMXY-SUMX*SUMY)**2
              DENOM=(N*SUMXSQ-SUMX**2) * (N*SUMYSQ-SUMY**2)
              RSQ=NUM/DENOM
              R=SQRT(RSQ)
              PRINT 800,R
      800     FORMAT(' CORRELATION COEFFICIENT = ',F6.2)
            ENDIF
            END
```

Figure 3-30. Program CORR Before Debugging

To execute the program, some test data is required. If possible, test cases for which results are known should be included. In the example in figure 3-30, the first test case consists of pairs of equal numbers; if the program is correct, it should calculate a correlation coefficient of 1.0.

The strategy for the first debug session is to allow the program to execute as far as possible, and to then use information obtained during this session to determine where to set traps and breakpoints for subsequent sessions.

The first debug session is shown in figure 3-31. No traps or breakpoints are set; GO is entered to initiate program execution, and the program is allowed to execute until termination, at which time the ABORT trap gives control to the user. The trap message indicates that an execution error has occurred. (Error 04 is caused by a computation involving an indefinite operand.)

Commands can now be entered to determine the cause of the errors. The PRINT command displays the contents of the array X, into which input values are stored. Note that only 5 values are displayed. (The X array is dimensioned 5.) This indicates a possible error because the input file contains ten sets of values. The next PRINT command, which uses an implied DO loop, indicates that the subscript might have exceeded the dimensioned size of X. Since the program contains no check on the number of records read, this permits an array bounds error to occur if the number of records exceeds the size of the array. When the program is corrected, a test on the number of records read will be included; but, in order to continue the debug session, the extra records are removed from the input file, and the program is rerun.

The second debug session is shown in figure 3-32. Abnormal termination occurs again at line 29. Since execution terminated within the loop, the values of I and N, along with some intermediate values, are printed. Although the data file contains only five records, the counter N has a value of 6. This causes another indexing error. By referring to the source listing, you can see that N is initialized to 1 and is incremented after each record is read; N will always contain a value of one greater than the actual number of records read.

The counter can be corrected by initializing it to 0 instead of 1. The CID output also shows that SUMYSQ contains an indefinite value. This is caused by failure to initialize SUMYSQ to 0. Without changing the source code and recompiling, debugging can continue by conducting another debug session and using assignment commands to insert the correct values for N and SUMYSQ.

Input Data:

```
1.0 1.0
10.0 10.0
7.6 7.6
2.9 2.9
5.1 5.1
3 3
100.5 100.5
7.0 7.0
```

Session Log:

```
CYBER INTERACTIVE DEBUG
?go ◄────────────────────────────────────── Initiate execution.

*T #18, ABORT CPU ERROR EXIT 04 IN L.29 ◄── Abort trap at line 29.
?print*,x ◄─────────────────────────────── Display input values.

 100.5 7. 7.6 2.9 5.1
?print*,(x(i),i=1,n) ◄──────────────────── This form of PRINT command
                                            indicates subscript error.
*WARN - SUBSCRIPT OUT OF RANGE
OK ?quit

  DEBUG TERMINATED
```

Figure 3-31. Input Data for First Test Case and Debug Session

```
CYBER INTERACTIVE DEBUG
?go

*T #18, ABORT CPU ERROR EXIT 04 IN L.29 ◄────────── ABORT trap at line 29.
?print*,n,i,sumx,sumy,sumxsq,sumysq,sumxy ◄──────── Display intermediate values.
 ─────────────────────────────────────────────────── N exceeds array boundary.
 6 1 1. 1. 1. I 0.
?quit        ─────────────────────────────────────── SUMYSQ contains meaningless value.

  DEBUG TERMINATED
```

Figure 3-32. Second Debug Session

The third debug session is shown in figure 3-33. When the breakpoint at line 22 occurs, N is set to 0 and SUMYSQ is set to 0.0. The GO command resumes execution; this time the program runs to completion. The value displayed for R, however, is clearly incorrect. (The correct value is known to be 1.0.) The next PRINT command shows that all the data values are being read correctly, and it is known from the second session that all the intermediate sums are correctly initialized. Another session will be conducted with execution suspended at various points within the computation portion of the program so that the progress of the calculations can be examined.

The fourth session is shown in figure 3-34. The correct initial values for N and SUMXY are inserted, as in the previous session.

The fourth debug session is shown in figure 3-34. Breakpoints are set at lines 22 and 35, and a STORE trap is set for the DO control variable I. The STORE trap is set to suspend execution on each pass through the loop of lines 25 through 31, when I is incremented. However, shortly after execution is initiated an ABORT trap occurs, indicating that the maximum time limit has been exceeded. The STORE trap activated interpret mode, causing the debug session to use too much time.

The session is terminated and a new one is initiated. This time, a breakpoint, instead of a STORE trap, is set at line 31. The breakpoint will have the same effect as the STORE trap; that is, execution will be suspended on each pass through the loop.

Execution is initiated and correct values are calculated for N and SUMXY as in the previous session. The display of intermediate values on each pass through the loop indicates a possible error: the value of the variable SUMXY should be increasing on each pass, as more values are added to it. However, the display shows that this value is not increasing. The calculation of SUMXY in line 30 is incorrect; the correct statement is:

```
SUMXY=SUMXY+X(I)*Y(I)
```

The debug session can be continued by using the assignment command to calculate and insert the correct value of SUMXY. First, execution is resumed to allow the loop to complete. After the last pass through the loop, the correct value is calculated by appropriate assignment commands.

The next suspension occurs at line 33. The value of RSQ is printed and is clearly wrong. (The correct value of R is known to be 1.0; therefore, the square of R should also be 1.0.) The next step is to examine the values used to calculate RSQ: NUM and DENOM. For RSQ to have a value of 1.0, NUM and DENOM must be equal. However, tha PRINT command shows that NUM and DENOM are not equal. NUM is implicitly an integer and, when the floating-point value was stored into NUM, truncation occurred. When the source program is corrected, the name NUM will be replaced by a name that is type REAL. The session can be continued, however, by once again using an assignment command to calculate the correct value and by substituting it for the incorrect value. This requires a temporary location into which the value of the numerator can be stored. The variable SUMX can be used for this temporary location since it is not referenced after line 35. After the numerator is calculated and stored in SUMX, an assignment command is used to calculate RSQ. The PRINT command shows that RSQ now has the correct value. Execution is resumed at line 35, which calculates the final result. The program runs to completion and tha session is terminated. The program now appears to execute correctly.

At this point, it is probably a good idea to incorporate all the accumulated changes into the source program, recompile, and rerun the program to verify the corrections. However, the program should not be considered completely debugged until it has been tested on additional sets of input data.

For the next test case, data records are included in which all the X values are equal. The input file and session log are shown in figure 3-35. The program runs to completion, but an error occurs in the SQRT routine and the indefinite character I is printed for the correlation coefficient. By

```
CYBER INTERACTIVE DEBUG
?set,breakpoint,l.22 <------------------- Set breakpoint at line 22.

?go

*B #1, AT L.22 <------------------------- Breakpoint suspends execution at line 22.
?print*,n,sumysq

 6 -I
?n=n-1         }
               }<------------------------ Calculate correct values for N and SUMYSQ.
?sumysq=0.0    }

?go

CORRELATION COEFFICIENT =    2.54 <------ Final result is incorrect.
*T #17, END IN L.39
?
       END  CORR
       25100 MAXIMUM EXECUTION FL.
        .327 CP SECONDS EXECUTION TIME.
quit

   DEBUG TERMINATED
```

Figure 3-33. Third Debug Session

```
CYBER INTERACTIVE DEBUG
?set,breakpoint,l.22
```

Set breakpoint at line 22.

```
?set,trap,store,i
```

Set STORE trap for variable I.

```
INTERPRET MODE TURNED ON
?set,breakpoint,l.35
```

Set breakpoint at line 35.

```
?go

*T #18, ABORT CP TIME LIMIT IN L.16
?quit
```

ABORT trap at line 16; time limit exceeded.

```
  DEBUG TERMINATED
lgo
```

Initiate new debug session.

```
CYBER INTERACTIVE DEBUG
?set,breakpoint,l.22

?set,breakpoint,l.31
```

Set breakpoint at line 31, instead of STORE trap.

```
?set,breakpoint,l.35

?go

*B #1, AT L.22
?n=n-1

?sumysq=0.0
```

Calculate correct values for N and SYMYSQ.

```
?go

*B #2, AT L.31
?print*,i,sumx,sumy,sumxsq,sumysq,sumxy

 1 1. 1. 1. 1. 2.
?go

*B #2, AT L.31
?print*,i,sumx,sumy,sumxsq,sumysq,sumxy

 2 11. 11. 101. 101. 20.
?go

*B #2, AT L.31
?print*,i,sumx,sumy,sumxsq,sumysq,sumxy

 3 18.6 18.6 158.76 158.76 15.2
?go

*B #2, AT L.31
?print*,i,sumx,sumy,sumxsq,sumysq,sumxy

 4 21.5 21.5 167.17 167.17 5.8
?go
```

Breakpoint suspends execution on each pass through loop; display intermediate values while execution is suspended.

```
*B #2, AT L.31
?print*,i

 5

?sumxy=x(1)*y(1)+x(2)*y(2)

?sumxy=sumxy+x(3)*y(3)+x(4)*y(4)

?sumxy=sumxy+x(5)*y(5)

?print*,sumxy
```

Calculate correct value for SUMXY.

```
 193.18
?go
```

Figure 3-34. Fourth Debug Session (Sheet 1 of 2)

```
*B #3, AT L.35
?print*,num,denom
                                                        Value of NUM is incorrect.
 66739 66739.555600002
?sumx=(n*sumxy-sumx*sumy)*(n*sumxy-sumx*sumy)  <--  Calculate correct value for
                                                     numerator, using SUMX for
?print*,sumx                                         temporary storage.

 66739.555600002  <--                               Calculate correct value
?rsq=sumx/denom                                      for RSQ.

?print*,rsq

 1.
?go

CORRELATION COEFFICIENT =     1.00  <--             Program prints correct result.
*T #17, END IN L.39
?
      END  CORR
      24600 MAXIMUM EXECUTION FL.
      3.140 CP SECONDS EXECUTION TIME.

quit

  DEBUG TERMINATED
```

Figure 3-34. Fourth Debug Session (Sheet 2 of 2)

Input Data:

```
3.0 1.0
3.0 5.1
3.0 7.6
3.0 10.0
```

Session Log:

```
CYBER INTERACTIVE DEBUG
?go

ARGUMENT INDEFINITE
FTN - INFORMATIVE ERROR NUMBER 39
TRACEBACK INITIATED BY SYSERR AT REL(ABS) ADDRESS 122(21407).  <--  System error messages.
CALLED BY SQRT AT ADDRESS 1(3574) WITH NO AP-LIST.
CORRELATION COEFFICIENT =      I  <--                     Final result is meaningless value.
*T #17, END IN L.40
?
      END  CORR
      24600 MAXIMUM EXECUTION FL.
       .149 CP SECONDS EXECUTION TIME.
print*,rsq,anum,denom
                                                         RSQ contains meaningless value.
 I 0. 0.
?quit

  DEBUG TERMINATED
```

Figure 3-35. Input Data for Second Test Case and Debug Session

using CID commands to display intermediate values, you can see that a division by zero has occurred. CID has helped determine the location of the error, but in order to understand why the error occurred, it is necessary to understand the mathematics of the program.

In the formula for the correlation coefficient, it can be shown that the calculation $n\Sigma x^2 - (\Sigma x)^2$ has a value of zero if all the x values are equal. Whenever a division occurs within a program, you should always be alert to the possibility of a zero denominator and include statements testing for that possibility.

To complete the debugging process, two more test cases are run: one in which the data correlates closely (figure 3-36), and one in which the values are widely scattered (figure 3-37). The results of both tests appear to be correct. In a real situation, correctness of the results should be verified whenever possible by comparing them with known results or by performing hand calculations. The final version of CORR, with all corrections included, is shown in figure 3-38.

Input Data:

```
10.1 10.1
20.5 20.5
6.0 6.0
34.0 32.9
4.4 4.5
```

Session Log:

```
CYBER INTERACTIVE DEBUG
?go

CORRELATION COEFFICIENT =    1.00
*T #17, END IN L.40
?
      END  CORR
      25100 MAXIMUM EXECUTION FL.
        .136 CP SECONDS EXECUTION TIME.
quit

  DEBUG TERMINATED
```

Figure 3-36. Input Data for Third Test Case and Debug Session

SAMPLE PROGRAM NEWT

Program NEWT finds a zero root of a function by Newton's method. Newton's method generates successive approximations to the equation f(x)=0 by applying the iteration:

$x_{i+1}=x_i-f(x_i)/d(x_i)$

where:

$f(x_i)$ is the current functional value.

$d(x_i)$ is the derivative of the current functional value.

x_i is the current approximation to the root.

x_{i+1} is the new approximation to the root.

Input Data:

```
0.0 0.0
0.0 100.0
0.0 0.0
0.0 500.0
0.1 10.0
```

Session Log:

```
CYBER INTERACTIVE DEBUG
?go

CORRELATION COEFFICIENT =     .29
      END  CORR
      25100 MAXIMUM EXECUTION FL.
        .142 CP SECONDS EXECUTION TIME.
*T #17, END IN L.40
?quit

  DEBUG TERMINATED
```

Figure 3-37. Input Data for Fourth Test Case and Debug Session

The program listing is shown in figure 3-39.

To use Newton's method, you start with an initial approximation and apply the preceding scheme to calculate a new and better approximation. You then substitute the new approximation into the relation and calculate a still closer approximation. Each successive approximation is closer to the desired root. The process is continued until the desired degree of accuracy is achieved.

The program to implement Newton's method consists of a main routine, a subroutine to apply Newton's method, and two function subprograms: F, which defines the function to be solved, and D, which calculates the derivative of the function.

The main program passes an initial approximation of the solution to subroutine NEWT, along with the function names. NEWT intializes an error flag IER and a variable ITS that contains the current number of iterations. The iterative scheme is applied in lines 400 through 480. If the initial approximation itself is a zero root, control returns to the main program. A zero derivative generates an error; therefore, a test is included for a zero value of the function D. Line 440 calculates a new approximation X. Line 410 tests the functional value FX of the current approximation; if FX has a value of zero, control returns to the main program. This type of test, as will be shown in the debug session, can lead to difficulty when used with an iterative scheme.

Subroutine NEWT returns the value of the solution (X), the number of iterations required (ITS), and the error flag (IER).

The function to be solved, defined in lines 550 through 590, is:

$f(x)=3.0x-(x+1.0)/(x-1.0)$

The derivative of the function, lines 610 through 670, is:

$d(x)=3.0+2.0/(x-1.0)^2$

```
PROGRAM CORR        74/74    OPT=0

      PROGRAM CORR
C  CORR CALCULATES A CORRELATION COEFFICIENT
      DIMENSION X(5),Y(5)
C
C...INITIALIZATION
C
      N=0†
      SUMX=0.0
      SUMY=0.0
      SUMXSQ=0.0
      SUMYSQ=0.0†
      SUMXY=0.0
C
C...READ NUMBERS TO BE CORRELATED
C
      OPEN (UNIT=2,FILE='CORFIL')
10    READ (2,*,END=20) X(N+1),Y(N+1)†
      N=N+1
      IF (N.GT.5) THEN†
        PRINT*, ' TOO MUCH INPUT. LIMIT IS 5 PAIRS'
      ELSE
        GO TO 10
      ENDIF
C
C...CALCULATE CORRELATION COEFFICIENT
C
20    IF (N.EQ.0) THEN
        PRINT*, ' EMPTY INPUT FILE'
      ELSE
        DO 30 I=1,N
        SUMX=SUMX+X(I)
        SUMY=SUMY+Y(I)
        SUMXSQ=SUMXSQ+X(I)**2
        SUMYSQ=SUMYSQ+Y(I)**2
        SUMYY=SUMXY+X(I)*Y(I)†
30      CONTINUE
        ANUM=(N*SUMXY-SUMX*SUMY)**2†
        DENOM=(N*SUMXSQ-SUMX**2) * (N*SUMYSQ-SUMY**2)
        IF (DENOM.EQ.0.0) THEN†
          PRINT*, ' BAD INPUT'
        ELSE
          RSQ=ANUM/DENOM†
          PRINT 800, R
800       FORMAT(' CORRELATION COEFFICIENT = ',F6.4)
        ENDIF
      ENDIF
      END
```

[†]Indicates correction.

Figure 3-38. Program CORR With Corrections

The debug session for subroutine NEWT is shown in figure 3-40. A STORE trap for the variable X would be useful since it would suspend execution after each new approximation is stored into X. However, the first attempt to set the STORE trap generates an error message: a subprogram argument cannot be referenced before the subprogram is called. The trap can be established by first setting a breakpoint at the beginning of the subroutine, then setting the trap while execution is suspended at that breakpoint.

The first SET,BREAKPOINT command generates an error message because line 380 is outside the home program and the program qualifier was omitted from the line number specification. The command is then reentered with the correct specification. When execution is suspended at line 380, the STORE trap is successfully established.

The first store into X occurs in line 400. The LIST VALUES command shows that all variables have been correctly initialized, except for the variables FX and DX which are not initialized until later in the program. Note that the parameters D and F are not displayed because they are external function names.

The next suspension is caused by an ABORT trap at line 410. This line contains a function reference. Error 00 indicates an illegal branch was attempted. (Error codes are described in appendix C.) An illegal branch can be caused by a missing subprogram or an incorrectly specified subprogram name. In this example, the error is in the CALL statement in the main program. The function name F is misspelled FF, and therefore is not passed to NEWT. The function reference resulted in an illegal branch.

```
00100 PROGRAM MAIN
00110 EXTERNAL F,D
00120 X0=0.0
00130 CALL NEWT(FF,D,X0,X,ITS,IER)
00140 IF(IER.NE.0) THEN
00150   PRINT*,' ERROR IN SUBROUTINE NEWT'
00160 ELSE
00170   PRINT 100, ITS,X
00180   100 FORMAT (' CONVERGENCE IN ',I4,' ITERATIONS. X= ',E12.4)
00190 ENDIF
00200 END
00210C
00220C   SUBROUTINE NEWT FINDS A ZERO ROOT OF AN EQUATION BY
00230C   NEWTONS METHOD
00240C
00250C   INPUT
00260C      F    NAME OF FUNCTION DEFINING EQUATION TO BE SOLVED
00270C      D    NAME OF FUNCTION DEFINING DERIVATIVE OF EQUATION
00280C      X0   INITIAL APPROXIMATION TO ROOT
00290C
00300C   OUTPUT
00310C      X    SOLUTION TO F(X)=0
00320C      ITS  NUMBER OF ITERATIONS REQUIRED FOR CONVERGENCE
00330C      IER  ERROR FLAG
00340C           0  NO ERRORS
00350C           1  ERRORS
00360C
00370 SUBROUTINE NEWT(F,D,X0,X,ITS,IER)
00380 IER=0
00390 ITS=0
00400 X=X0
00410 10 FX=F(X)
00420 IF (FX.EQ.0.0) RETURN
00430 DX=D(X)
00440 IF (DX.EQ.0.0) THEN
00450   PRINT '('' DERIV= 0 AT '',F6.2,'' SPECIFY DIFFERENT X0'')'
00460   IER=1
00470   RETURN
00480 ELSE
00490   X=X-FX/DX
00500   ITS=ITS+1
00510   GO TO 10
00520 ENDIF
00530 END
00540C
00550C   F DEFINES A FUNCTION TO BE SOLVED BY NEWTONS METHOD
00560C
00570 FUNCTION F(X)
00580 IF(X.EQ.1.0) STOP 'BAD ARG TO F'
00590 F=3.0*X-(X+1.0)/(X-1.0)
00600 RETURN
00610 END
00620C
00630C   D CALCULATES THE DERIVATIVE OF F
00640C
00650 FUNCTION D(X)
00660 IF(X.EQ.1.0) STOP 'BAD ARG TO D'
00670 D=3.0+2.0/(X+1.0)**2
00680 RETURN
00690 END
```

Figure 3-39. Subroutine NEWT and Main Program Before Debugging

The debug session is continued by using an assignment command to calculate the correct value for FX. Execution is then resumed at line 420.

Note that when execution is resumed after an ABORT trap, a line number must be specified in the GO command to avoid executing the system error code.

The next suspension occurs when a new value is stored into X in line 490. The value is printed and execution is resumed. On the next iteration, the ABORT trap again occurs at line 410. A new value for FX is calculated and displayed. The STORE trap is removed to avoid unnecessary suspensions. Execution is again resumed at line 420.

```
CYBER INTERACTIVE DEBUG
? set,trap,store,p.newt_x            <-- Attempt to set STORE trap fails.
 *ERROR - PARAMETER REFERENCED BEFORE FIRST SUBROUTINE CALL
? set,breakpoint,l.380               <-- Attempt to set breakpoint fails.
 *WARN - LINE 380 NOT EXECUTABLE - LINE 200 WILL BE USED
 OK ? set,breakpoint,p.newt_l.380    <-- Set breakpoint at line 380 of NEWT.
? go
 *B #1, AT P.NEWT_L.380              <-- Breakpoint detected at line 380 of NEWT.
? set,trap,store,x                   <-- Set STORE trap for variable X.
 INTERPRET MODE TURNED ON
? go
 *T #1, STORE INTO X (OF P.MAIN) IN L.400  <-- STORE trap suspends execution in line 400.
? list,values,p.newt
 P.NEWT
 DX = -I,   FX = -I,   IER = 0,   ITS = 0,   X = 0.0,   X0 = 0.0
? go
 *T #18, ABORT CPU ERROR EXIT 00 IN L.410  <-- ABORT trap in line 410.
? fx=3.0*x-(x+1.0)/(x-1.0)           <-- Calculate correct value of FX.
? print*,fx
  1.
? go,l.420                           <-- Resume execution at line 420.
 *T #1, STORE INTO X (OF P.MAIN) IN L.490  <-- STORE trap suspends execution in line 490.
? print*,x
  -.2
? go
 *T #18, ABORT CPU ERROR EXIT 00 IN L.410  <-- ABORT trap suspends execution in line 410.
? fx=3.0*x-(x+1.0)/(x-1.0)           <-- Calculate new value for FX.
? print*,fx
  6.6666666666666E-02
? clear,trap,store                   <-- Remove STORE trap.
 INTERPRET MODE TURNED OFF
? go,l.420
 *T #18, ABORT CPU ERROR EXIT 00 IN L.410
? fx=3.0*x-(x+1.0)/(x-1.0)
? print*,fx
  1.9032332033941E-02
? go,l.420
 *T #18, ABORT CPU ERROR EXIT 00 IN L.410
? fx=3.0*x-(x+1.0)/(x-1.0)
? print*,fx
  5.6719259568219E-03
? go,l.420
 *T #18, ABORT CPU ERROR EXIT 00 IN L.410
? fx=3.0*x-(x+1.0)/(x-1.0)
? print*,fx
  1.7103829132736E-03
? go,l.420
 *T #18, ABORT CPU ERROR EXIT 00 IN L.410
? fx=3.0*x-(x+1.0)/(x-1.0)
? print*,fx,its                      <-- ITS contains the current number of
  5.1756717224194E-04 5                  iterations.
? quit                               <-- Terminate session.

SRU     14.185 UNTS.

 RUN COMPLETE.
```

ABORT trap occurs on each pass through loop. Each time execution is suspended, calculate a new value for FX, display the value, and resume execution.

Figure 3-40. Debug Session for Subroutine NEWT

The process of calculating a value for FX and resuming execution at line 420 is repeated for subsequent iterations, until the value of FX appears to be converging to a solution.

After a few passes through the loop, the solution appears to be correctly converging to zero. However, the test to exit from the loop is satisfied only if FX is equal to zero. If FX does not eventually become equal to zero, an infinite loop results. For this reason, you should avoid testing a calculated value for strict equality.

To prevent an infinite loop, the test for convergence must be changed to exit on a sufficiently small value of FX. The constant used for the test depends on the desired degree of accuracy; for example, for 3-place accuracy, a value of .0001 is used. A limit should also be imposed on the number of passes through the loop, since the method might not converge for certain functions. The loop can be replaced with a DO loop with an arbitrary limit of 100 passes. The corrected version of the program is shown in figure 3-41.

```
PROGRAM MAIN
EXTERNAL F,D
X0=0.0
CALL NEWT(F,D,X0,X,ITS,IER)†
IF(IER.NE.0) THEN
  PRINT*,' ERROR IN SUBROUTINE NEWT'
ELSE
  PRINT 100, ITS,X
  100 FORMAT (' CONVERGENCE IN ',I4,' ITERATIONS. X= ',E12.4)
ENDIF
END
C
C   SUBROUTINE NEWT FINDS A ZERO ROOT OF AN EQUATION BY
C   NEWTONS METHOD
C
C   INPUT
C      F     NAME OF FUNCTION DEFINING EQUATION TO BE SOLVED
C      D     NAME OF FUNCTION DEFINING DERIVATIVE OF EQUATION
C      X0    INITIAL APPROXIMATION TO ROOT
C
C   OUTPUT
C      X     SOLUTION TO F(X)=0
C      ITS   NUMBER OF ITERATIONS REQUIRED FOR CONVERGENCE
C      IER   ERROR FLAG
C            0  NO ERRORS
C            1  ERRORS
C
SUBROUTINE NEWT(F,D,X0,X,ITS,IER)
IER=0
ITS=0
EPS=0.0001†
X=X0
C
C     ITERATE TO FIND ROOT
C
DO 10 I=1,100†
FX=F(X)
IF (FX.LE.EPS) RETURN†
DX=D(X)
IF (DX.EQ.0.0) THEN
  PRINT '('' DERIV= 0 AT '',F6.2,'' SPECIFY DIFFERENT X0'')'
  IER=1
  RETURN
ELSE
  X=X-FX/DX
  ITS=ITS+1
ENDIF
10 CONTINUE
PRINT*,' METHOD HAS NOT CONVERGED IN 100 ITERATIONS'†
IER=1
RETURN
END
C
C   F DEFINES A FUNCTION TO BE SOLVED BY NEWTONS METHOD
C
FUNCTION F(X)
IF(X.EQ.1.0) STOP 'BAD ARG TO F'
F=3.0*X-(X+1.0)/(X-1.0)
RETURN
END
C
C   D CALCULATES THE DERIVATIVE OF F
C
FUNCTION D(X)
IF(X.EQ.1.0) STOP 'BAD ARG TO D'
D=3.0+2.0/(X+1.0)**2
RETURN
END
```

†Indicates corrections.

Figure 3-41. Subroutine NEWT and Main Program With Corrections

AUTOMATIC EXECUTION OF CID COMMANDS 4

In many cases, you will find it necessary to enter the same command or sequence of commands repeatedly during the course of a debug session. This type of situation is illustrated in the debugging examples of section 3. In the debug session for subroutine NEWT (figure 3-40 in section 3), the same sequence of commands is needed on each pass through the DO loop. Debugging program CORR required several debug sessions (figures 3-31 through 3-37 in section 3). During each session, it was necessary to reenter the assignment commands that calculated the correct intermediate values. Reentering complicated assignment commands in this manner can be a time-consuming process.

To eliminate the need for repeatedly entering sequences of commands, CYBER Interactive Debug (CID) provides the ability to define, save, and automatically execute sequences of commands. This feature can be used to improve debugging efficiency whenever the same group of CID commands must be entered repeatedly. Automatic command execution is commonly used when debugging DO loops and frequently-called subroutines, and in multiple debug sessions that require the same commands. In addition, CID provides some special sequence commands that allow you to incorporate FORTRAN-like logic into command sequences. For example, sequence commands allow branching and conditional execution of CID commands.

COMMAND SEQUENCES

A command sequence is a series of CID commands which is to be executed automatically either when certain conditions occur or when you enter the appropriate command from the terminal.

There are three ways to establish a command sequence:

- By defining a command sequence as part of a trap or breakpoint. This causes the sequence to be executed whenever the trap or breakpoint occurs. A sequence defined in this manner is called a trap body or breakpoint body.

- By defining a command sequence called a group. A group is executed by issuing a READ command from the terminal or from another command sequence.

- By creating a file, outside of CID, which contains a sequence of CID commands. The commands in this file are executed by issuing a READ command at the terminal, or from another command sequence.

During normal execution, CID prompts for user input after a command is executed. During sequence execution, however, CID executes all the commands in the sequence without interruption. Once execution of the sequence is completed, execution of your program resumes at the point where it was suspended. You do not get control during sequence execution unless you provide for it using the PAUSE command, described later in this section.

Command sequences can be nested; that is, command sequences can be called from other command sequences.

COLLECT MODE

Collect mode is a mode of execution in which CID commands are not executed immediately after they are entered, but are included in a command sequence for execution at a later time. To define a trap body, breakpoint body, or command group, you must first activate collect mode. The procedure for entering and leaving collect mode is described under Traps and Breakpoints With Bodies.

Commands in a sequence that you are creating cannot be altered while CID is in collect mode. If you make a mistake or wish to change a command that you have entered, you must leave collect mode and proceed as described under Editing a Command Sequence.

MULTIPLE COMMAND ENTRY

You can enter more than one command on a single line, but not exceeding 150 six bit characters in a line. The commands are separated with a semicolon, as in the following example:

```
PRINT*,X;SET,BREAKPOINT,L.25;GO
```

In interactive mode, CID does not execute the commands until you press the RETURN key; it then executes the commands in the order you entered them. In collect mode, the commands are not executed, but are included in the command sequence.

This method of command entry is especially convenient when you are defining command sequences because you don't have to wait for an input prompt before entering each command. This technique is illustrated later in this section.

SEQUENCE COMMANDS

CID provides a set of commands intended specifically for use with command sequences. These commands are summarized in table 4-1.

TRAPS AND BREAKPOINTS WITH BODIES

A body is a sequence of commands specified as part of a SET,TRAP or SET,BREAKPOINT command. To define a trap or breakpoint with a body, you must first initiate collect mode by including a left bracket ( ) as the last parameter of the SET,TRAP or SET,BREAKPOINT command. For example:

```
SET,TRAP,LINE,P.MAIN
```

The bracket and the preceding parameter must not be separated by a comma; the blank separator is optional.

When the above command is entered, CID displays the message and prompt:

```
IN COLLECT MODE
?
```

TABLE 4-1. SEQUENCE COMMANDS

Command	Description
PAUSE	Temporarily suspends execution of the current command sequence and reinstates interactive mode allowing commands to be entered from the terminal.
MESSAGE	Displays a message at the terminal.
GO	Resumes the process most recently suspended.
EXECUTE	Resumes execution of the user program.
IF	Performs conditional execution of commands.
LABEL	Defines a label within a command sequence.
JUMP	Transfers control within a command sequence to a label defined by a LABEL command.
READ	Initiates execution of a command sequence defined as a group or stored on a file; reestablishes trap, break point, and group definitions stored on a file.

You then enter the commands that are to comprise the body. Each command entered while CID is in collect mode becomes part of the body. CID scans the command for errors but does not execute the command. You can include any number of commands in a body, although command sequences should be kept short and simple.

To leave collect mode and return to interactive mode, enter a right bracket (]) in response to the ? prompt or at the end of a command line. CID then displays:

```
END COLLECT MODE
?
```

and you can continue the session.

An example of a breakpoint definition with a body is as follows:

```
SET,BREAKPOINT,L.8[
X=0.0
Y=Y+1.0
PRINT*,X,Y
]
```

Note that this command sequence can also be entered as follows:

```
SET,BREAKPOINT,L.8 [X=0.0;Y=Y+1.0;PRINT*,X,Y]
```

When a trap or breakpoint with a body is encountered, program execution is suspended and the commands in the body are executed automatically. Program execution then resumes at the trap or breakpoint location; CID does not give control to you upon completion of the sequence.

When a trap or breakpoint with a body is encountered during execution, the normal trap or breakpoint message is not displayed. However, you can provide your own notification of the execution of a trap or breakpoint body by including a MESSAGE command (table 4-1) in the sequence. The format of the MESSAGE command is:

```
MESSAGE,'character string'
```

When a MESSAGE command is encountered, the character string is displayed, and execution of the sequence continues. (You do not get control after execution of a MESSAGE command).

You do not receive control during execution of a sequence unless you have provided for it by including a PAUSE command (described under Receiving Control During Sequence Execution) in the body. When the body has been executed, execution of your program automatically resumes at the location where it was suspended.

An example of the procedure for establishing a breakpoint body is illustrated in figure 4-1. The program used in this example is shown in figure 3-4 in section 3. A breakpoint is established at the RETURN statement in subroutine AREA. The breakpoint body contains the following commands:

- A MESSAGE command to display a message at the beginning of the sequence.
- A DISPLAY command to display the contents of the #LINE variable which contains the current line number.
- A PRINT command to display the input values and the value of A.

Subroutine AREA is called four times; each time the breakpoint is detected, the commands in the sequence are executed.

DISPLAYING TRAP AND BREAKPOINT BODIES

You can display a list of the commands in a breakpoint body by specifying the breakpoint location in the following LIST,BREAKPOINT commands:

LIST,BREAKPOINT,loc_1,loc_2,...,loc_n

Displays the complete definitions, including the bodies (if any), of the breakpoints at statements loc_1, loc_2,...; loc_i has one of the forms S.n, L.n, P.prog_S.n, or P.prog_L.n.

LIST,BREAKPOINT,#n_1,#n_2,...,#n_m

Displays the complete definitions, the bodies (if any), of the breakpoints having numbers n_1, n_2,...,n_m; breakpoint numbers are assigned by CID when the breakpoints are established.

Other forms of the LIST,BREAKPOINT command list the breakpoint location but not the commands in the body.

Examples:

```
LIST,BREAKPOINT,#1,#5,#6

LIST,BREAKPOINT,P.SUBC_L.10
```

To display a list of the commands in a trap body, enter the following form of the LIST,TRAP command:

LIST,TRAP,#n_1,#n_2,...,#n_m

Displays the bodies (if any) of the traps having numbers n_1, n_2,...,n_m; the trap numbers are assigned by CID when the traps are established.

This command displays the trap type, location, and body. Other forms of the LIST,TRAP command list only the trap type and location.

Example:

LIST,TRAP,#2,#5

lists the type, location, and body of the traps numbered 2 and 3.

Figure 4-2 illustrates a LIST,BREAKPOINT command for the breakpoint established in figure 4-1.

```
CYBER INTERACTIVE DEBUG
?set,breakpoint,p.area_l.7 [     <-- Set breakpoint at line 7 of subroutine AREA
                                     and activate collect mode.
IN COLLECT MODE
?message,'in subroutine area'

?display,#line                   <-- Breakpoint body. Commands are included
                                     to display message, current FORTRAN line
?print*,'input is ',x1,y1,x2,y2,x3,y3    number, input values, and final result.

?print*,'area is ',a

?]                               <-- Turn off collect mode.

END COLLECT
?go                              <-- Initiate program execution.

IN SUBROUTINE AREA
#LINE = P.AREA_L.7
 INPUT IS 0. 0. 2. 0. 0. 2.
 AREA IS 2.
IN SUBROUTINE AREA
#LINE = P.AREA_L.7
 INPUT IS 0. 1.  .5 2. -1. 1.2   <-- On each pass through subroutine AREA,
 AREA IS  .55                        the breakpoint suspends program execution
IN SUBROUTINE AREA                   and the commands in the body are
#LINE = P.AREA_L.7                   executed. After all input records are
 INPUT IS 6.1 2.  .1 -4. 3.2 7.      processed, the program terminates.
 AREA IS 23.7
IN SUBROUTINE AREA
#LINE = P.AREA_L.7
 INPUT IS  .2 -2.9 -1.3 8. 5.6 2.8
 AREA IS 33.705
*T #17, END IN P.RDTR_L.7
?
     STOP
      20100 MAXIMUM EXECUTION FL.
      1.672 CP SECONDS EXECUTION TIME.
quit

  DEBUG TERMINATED
```

Figure 4-1. Debug Session Illustrating Breakpoint With Body

```
?list,breakpoint,#1   <-- Display the complete definition of breakpoint #1, including the
                          breakpoint location and all commands in the breakpoint body.
*B #1 = P.AREA_L.7
SET,BREAKPOINT,P.AREA_L.7 [
MESSAGE,'IN SUBROUTINE AREA'
DISPLAY,#LINE
PRINT*,'INPUT IS ',X1,Y1,X2,Y2,X3,Y3
PRINT*,'AREA IS ',A
]
?
```

Figure 4-2. Debug Session Illustrating LIST,BREAKPOINT Command for Breakpoint With Body

GROUPS

A group is a sequence of commands established and assigned a name during a debug session, but not explicitly associated with a trap or breakpoint. A group exists for the duration of the session and is executed by entering an appropriate READ command. The command to establish a group is:

```
SET,GROUP,name[
```

where name is a name by which you will reference the group. The left bracket activates collect mode, as with trap and breakpoint bodies. Any number of CID commands subsequently entered become part of the sequence until you terminate the sequence by entering a right bracket.

The command to execute a group is:

```
READ,name
```

where name is the group name assigned in the SET,GROUP command. You can issue a READ command directly from the terminal or from another command sequence. In response to a READ command, CID executes the commands in the group. After a group has been executed, control returns to CID (if the READ was entered from the terminal) or to the next command in the sequence that issued the READ.

A group can be used when the same sequence of commands is to be executed at different locations in a program. A trap or breakpoint body is executed only when the trap or breakpoint occurs, but a group can be executed at any time. Following is an example of a simple group definition:

```
SET,GROUP,GRPA[
X=Y+Z
PRINT*,X,Y,Z
]
```

This command sequence is executed by entering the command:

```
READ,GRPA
```

When a group is established, it is assigned a number in the same manner as traps and breakpoints. You can refer to a group by number or by name in the LIST, CLEAR, and SAVE commands.

You can list the commands comprising a group with the following commands:

LIST,GROUP

Lists the names and numbers of all groups defined for the current debug session; does not list the commands contained in the groups.

LIST,GROUP,$name_1$,$name_2$,...,$name_n$

Lists the commands contained in the specified groups.

LIST,GROUP,#n_1,#n_2,...,#n_m

Lists the commands contained in the groups identified by the specified numbers.

Note that the first command form lists only the names and numbers of groups, whereas the second and third forms list the commands comprising the specified groups.

Normally, a group exists for the duration of a debug session. You can remove existing groups from the current debug session by entering one of the following commands:

CLEAR,GROUP

Removes all currently-defined groups.

CLEAR,GROUP,$name_1$,$name_2$,...,$name_n$

Removes the specified groups.

CLEAR,GROUP,#n_1,#n_2,...,#n_m

Removes the groups identified by the specified numbers.

Figures 4-3 and 4-4 illustrate debug sessions using groups. In figure 4-3, two breakpoints are set in subroutine SETB. When either breakpoint is reached, the READ command is issued from the terminal. In figure 4-4, the same breakpoints are established, except that a body containing a READ command is defined for each breakpoint. This causes the body to be executed automatically when the breakpoints are encountered, with no intervention from the user. By defining a single group instead of a body for each breakpoint, it is necessary to enter the command sequence only once. The group is listed with the LIST,GROUP command.

In figure 4-4, note that there are three levels of execution: the program, the breakpoint body, and the group. When the breakpoint is reached, the program is suspended, and execution of the breakpoint body is initiated. When the READ command is encountered, execution of the breakpoint body is suspended while the group is executed. When execution of the group is complete, execution of the suspended breakpoint body resumes at the command following the READ. When execution of the breakpoint body is complete, execution of the suspended program resumes.

Groups are especially useful when the same sequence of commands is to be executed at more than one location within a program. An example of this is illustrated in figure 4-5. The program MATOP defines two matrices and calls subroutines to add, subtract, and multiply the matrices and store the results in an array called MWRK. The purpose of the debug session is to print the contents of MWRK after each subroutine call. To accomplish this, a group named PRNT is defined to contain appropriate PRINT commands. After each subroutine call, a breakpoint is set with a body containing a command to execute the commands in group PRNT. When each breakpoint is encountered, the group commands are automatically read and executed. The debug session in figure 4-6 is identical to figure 4-5 except that the command READ,PRNT is issued from the terminal instead of a breakpoint body. Note that when the READ command is executed in the breakpoint body, program execution continues after the group commands are executed. When the READ command is entered at the terminal, control returns to CID after the group commands are executed, and program execution must be resumed with a GO command.

```
CYBER INTERACTIVE DEBUG
?set,group,grpa [

IN COLLECT MODE
?message,'executing grpa'

?display,#line

?print*,'k= ',k,' b= ',b

?]

END COLLECT
?set,breakpoint,p.setb_l.10

?go

*B #1, AT P.SETB_L.10
?read,grpa

EXECUTING GRPA
#LINE = P.SETB_L.10
 K= 1  B= -1. -1. -1. -1. -1.
?go

*B #1, AT P.SETB_L.10
?read,grpa

EXECUTING GRPA
#LINE = P.SETB_L.10
 K= 2  B= 1. 1. 1. 1. 1.
?quit

  DEBUG TERMINATED
```

Define group GRPA. Commands are included to display a message, the current line number, and the values of the variable K and the array B.

Set breakpoint at line 10 of subroutine SETB.

Initiate program execution.

Breakpoint suspends execution at line 10 of SETB.

Execute the commands in GRPA.

Resume program execution.

Figure 4-3. Debug Session Illustrating Group Execution Initiated at Terminal

ERROR PROCESSING DURING SEQUENCE EXECUTION

When CID is in collect mode and you are defining a command sequence, CID scans each command you enter for syntactic errors. If a syntactic error is detected, CID displays an error message followed by a ? prompt, after which you can reenter the command. Other errors, however, such as nonexistent line number or variable name, cannot be detected until CID attempts to execute the command.

CID issues normal error and warning messages during sequence execution. When an error or warning condition is detected, CID suspends execution of the sequence and issues a message followed by an input prompt (? for error messages; OK? for warning messages) on the next line. You then can instruct CID to disregard the command, replace the command with another command, or, in the case of warning messages, execute the command. The ways in which you can respond to error and warning messages are summarized as follows:

User Response	CID Action
OK or YES	For warning messages only, execute the command.
NO	Disregard the command. Execution resumes at the next command in the sequence.
NO,SEQ	Disregard the command and all remaining commands in the sequence.
Any CID command	Execute the specified command line in place of the current command, and resume execution of the sequence.

An example of error processing during sequence execution is illustrated in figure 4-7. During execution of group CGR, CID issues a warning message and two error messages. After each message is issued, CID gives control to the user. In response to the warning message, the user enters a new command to be executed in place of the incorrect command. In response to the first error message, the user enters NO, instructing CID to ignore the incorrect command and resume execution of the sequence. In response to the second error message, the user enters NO,SEQ, instructing CID to disregard the incorrect command and all remaining commands in the sequence and to give control to the user.

RECEIVING CONTROL DURING SEQUENCE EXECUTION

Normally, a command sequence executes to completion without returning control to CID. There might be instances, however, when you would like to temporarily gain control during execution of a sequence for the purpose of entering other commands. You can do this by using the PAUSE command.

```
CYBER INTERACTIVE DEBUG
?set,group,grpa [

IN COLLECT MODE
?message,'executing grpa'

?display,#line

?print*,'k= ',k,' b= ',b

?]

END COLLECT
?set,breakpoint,l.6 [

IN COLLECT MODE
?read,grpa

?]

END COLLECT
?set,breakpoint,l.8 [

IN COLLECT MODE
?read,grpa

?]

END COLLECT
?list,breakpoint,#1,#2

*B #1 = L.6
SET,BREAKPOINT,L.6 [
READ,GRPA
]
*B #2 = L.8
SET,BREAKPOINT,L.8 [
READ,GRPA
]
?go

EXECUTING GRPA
#LINE = P.PROGY_L.6
 K= 1  B= -1. -1. -1. -1. -1.
EXECUTING GRPA
#LINE = P.PROGY_L.8
 K= 2  B= 1. 1. 1. 1. 1.
*T #17, END IN L.8
?
      END  PROGY
      12300 MAXIMUM EXECUTION FL.
       .601 CP SECONDS EXECUTION TIME.
quit

  DEBUG TERMINATED
```

Define group GRPA. Commands are included to display a message, the current FORTRAN line number, and the values of variables K and B.

Set breakpoints at lines 6 and 8 of home program. Define a body for each breakpoint which contains a READ command to initiate execution of the commands in GRPA.

List the definitions of the breakpoints.

Initiate program execution.

The breakpoints of lines 6 and 8 suspend program execution and the READ commands in the bodies are automatically executed, causing the commands in GRPA to be executed.

Figure 4-4. Debug Session Illustrating Group Execution Initiated From Breakpoint Body

```
         PROGRAM MATOP        74/74    OPT=0

 1                    PROGRAM MATOP
 2                    DIMENSION MAT1(3,3),MAT2(3,3),MWRK(3,3)
 3                    DATA MAT1/2,6,4,3,8,9,7,5,8/,
 4                   1     MAT2/1,0,0,0,1,0,0,0,1/
 5                    N=3
 6                    CALL MATADD(N,MAT1,MAT2,MWRK)
 7                    CALL MATSUB(N,MAT1,MAT2,MWRK)
 8                    CALL MATMPY(N,MAT1,MAT2,MWRK)
 9                    END
```

```
CYBER INTERACTIVE DEBUG
?set,group,prnt [

IN COLLECT MODE
?message,'contents of mwrk'

?display,#line

?print*,mwrk(1,1),mwrk(1,2),mwrk(1,3)

?print*,mwrk(2,1),mwrk(2,2),mwrk(2,3)

?print*,mwrk(3,1),mwrk(3,2),mwrk(3,3)

?]
```

Define group PRNT. Commands are included to display the values of array MWRK.

```
END COLLECT
?list,group,prnt
```

Display the definition of group PRNT.

```
*G #1 = PRNT
SET,GROUP,PRNT [
MESSAGE,'CONTENTS OF MWRK'
DISPLAY,#LINE
PRINT*,MWRK(1,1),MWRK(1,2),MWRK(1,3)
PRINT*,MWRK(2,1),MWRK(2,2),MWRK(2,3)
PRINT*,MWRK(3,1),MWRK(3,2),MWRK(3,3)
]
?set,breakpoint,l.7 [

IN COLLECT MODE
?read,prnt

?]

END COLLECT
?set,breakpoint,l.8 [

IN COLLECT MODE
?read,prnt

?]

END COLLECT
?set,breakpoint,l.9 [

IN COLLECT MODE
?read,prnt

?]
```

Set breakpoints at lines 7, 8, and 9. In each breakpoint body, include a READ command to initiate execution of the commands in group PRNT.

Figure 4-5. Program MATOP and Debug Session (Sheet 1 of 2)

```
END COLLECT
?go                                 <---- Initiate program execution.

CONTENTS OF MWRK       )
#LINE = P.MATOP_L.7    )            Breakpoint suspends execution at line 7.  READ command is
 3 3 7                 ) <--------- executed, and control returns to program.
 6 9 5                 )
 4 9 9                 )
CONTENTS OF MWRK       )
#LINE = P.MATOP_L.8    )            Breakpoint suspends execution at line 8.  READ command is
 1 3 7                 ) <--------- executed.
 6 7 5                 )
 4 9 7                 )
CONTENTS OF MWRK       )
#LINE = P.MATOP_L.9    )            Breakpoint suspends execution at line 9.  READ command is
 2 3 7                 ) <--------- executed, and program runs to completion.
 6 8 5                 )
 4 9 8                 )
*T #17, END IN L.9
?
      END  MATOP
      12600 MAXIMUM EXECUTION FL.
      2.020 CP SECONDS EXECUTION TIME.
quit

  DEBUG TERMINATED
```

Figure 4-5. Program MATOP and Debug Session (Sheet 2 of 2)

PAUSE COMMAND

The purpose of the PAUSE command is to suspend execution of a command sequence. The formats of the PAUSE command are:

PAUSE

PAUSE,'string'

where string is any string of characters. When CID encounters this command in a sequence, execution of the sequence is suspended and CID gets control, allowing you to enter commands. If string is specified, the character string is displayed when the PAUSE command is executed.

The PAUSE command is valid only in a command sequence; it cannot be entered directly from the terminal.

When a PAUSE command is encountered in a trap or breakpoint body, CID displays the trap or breakpoint message followed by any message included in the PAUSE command, and prompts for user input.

Execution of the suspended sequence can be resumed by either the GO or the EXECUTE command. These commands are explained in the following paragraphs.

GO AND EXECUTE COMMANDS

The functions of the GO and EXECUTE commands are identical except when issued following suspension of a command sequence. When program execution has been suspended by a trap or breakpoint, both commands resume program execution. However, when execution of a command sequence has been suspended, the GO and EXECUTE commands differ as follows:

- GO resumes execution of the suspended sequence.
- EXECUTE causes an immediate exit from the sequence and resumes execution of the program.

The debug session in figure 4-8 illustrates the PAUSE and GO commands. This session was produced by executing program AREA, shown in figure 3-4 in section 3, under CID control. The purpose of this session is to suspend execution at the beginning of the subroutine in order to display the input values, to change them if necessary, and to suspend execution at the end of the subroutine in order to display the calculated area. To accomplish this, a breakpoint with a body is set at line 2 of subroutine AREA. Two commands are included in the body: a PRINT command and a PAUSE command. The PRINT command displays the input values, and the PAUSE command suspends execution of the sequence so that the user can change these values. A STORE trap is then established for the variable A. A body containing a command to print the value of A is defined for this trap. On each of the three passes through subroutine AREA, the commands in the sequence are executed automatically. When the PAUSE command is detected on the first pass, the user enters GO to resume sequence execution. (In this case, GO and EXECUTE have the same effect since PAUSE is the last command in the sequence.) On the next two passes through the subroutine, assignment commands are entered to change the values of some of the input variables while sequence execution is suspended because of the PAUSE command.

```
CYBER INTERACTIVE DEBUG
?set,group,prnt [

IN COLLECT MODE
?message,'contents of mwrk'

?display,#line

?print*,mwrk(1,1),mwrk(1,2),mwrk(1,3)      Define a group to print the values of array MWRK.

?print*,mwrk(2,1),mwrk(2,2),mwrk(2,3)

?print*,mwrk(3,1),mwrk(3,2),mwrk(3,3)

?]

END COLLECT
?set,breakpoint,l.7

?set,breakpoint,l.8      Set breakpoints at lines 7, 8, and 9.

?set,breakpoint,l.9

?go

*B #1, AT L.7
?read,prnt      Initiate execution of group while execution is suspended at line 7.

CONTENTS OF MWRK
#LINE = P.MATOP_L.7
 3 3 7
 6 9 5
 4 9 9
?go      Resume program execution.

*B #2, AT L.8
?read,prnt      Initiate execution of group while execution is suspended at line 8.

CONTENTS OF MWRK
#LINE = P.MATOP_L.8
 1 3 7
 6 7 5
 4 9 7
?go      Resume program execution.

*B #3, AT L.9
?read,prnt      Initiate execution of group while execution is suspended at line 9.

CONTENTS OF MWRK
#LINE = P.MATOP_L.9
 2 3 7
 6 8 5
 4 9 8
?quit

  DEBUG TERMINATED
```

Figure 4-6. Second Debug Session for Program MATOP

```
?list,group,cgr ←——————— Display group definition.

*G #1 = CGR
SET,GROUP,CGR [
PRINT*,(A(I),I=1,50)
X=1.0
C=2.0
Z=X+Y
PRINT*,'Z= ',Z
A(1)=B+C
PRINT*,'A(1)= ',A(1)
]
?read,cgr ←——————— Initiate execution of group CGR.
                                                                   Indicated command
*CMD - ( PRINT*,(A(I),I=1,50) )  *WARN - SUBSCRIPT OUT OF RANGE ←—— contains an error.
OK ?print*,(a(i),i=1,5)←—— Replace incorrect command with new command and
                           resume group execution.
 1. 2. 3. 4. 5.
*CMD - ( C=2.0 )  *ERROR - NO PROGRAM VARIABLE C ←—— Indicated command contains an error.
?no ←——————— Disregard erroneous command and resume group
             execution.
 Z= 1.
*CMD - ( A(1)=B+C )  *ERROR - NO PROGRAM VARIABLE C ←— Indicated command contains an error.
?no,seq ←——————— Disregard erroneous command and all remaining
                 commands in group. Control returns to CID.
?go ←——————— Resume program execution.

*T #17, END IN L.5
?
```

Figure 4-7. Debug Session Illustrating Error Processing During Sequence Execution

Both the GO and EXECUTE commands can be used to resume program execution at a location other than the one where execution was suspended. The command forms are:

GO,loc

EXECUTE,loc

where loc is a specification of the form L.n or S.n. These command forms resume execution at the specified statement.

The GO and EXECUTE commands can be used to skip sections of code, as illustrated in figure 4-9. In this example, the main program passes two values, A and B, to a subroutine which calculates a value for C. C is then used in a subsequent calculation. The user wishes to skip the call to SUB, assigning instead his own value to C. A breakpoint is set at line 4 to suspend execution immediately before execution of the CALL statement. When execution is suspended at the breakpoint location, a value is assigned to C. The user then enters GO,L.5 to resume execution at line 5, and line 4 is bypassed.

CONDITIONAL EXECUTION OF CID COMMANDS

CID allows conditional execution of commands in much the same manner as FORTRAN allows for executable statements. CID provides an IF command that is similar to the FORTRAN IF statement and a JUMP command that is similar to the FORTRAN GO TO statement.

IF COMMAND

The format of the IF command is:

IF (expr) command

where expr is a relational expression and command is any valid CID command. If the relational expression is true, CID executes the command.

The form of a relational expression is the same as in FORTRAN. The following relational operators are valid:

.EQ. .GT.
.NE. .LE.
.LT. .GE.

The following restrictions apply to the IF command:

- Only variables defined in the current home program can appear.
- CID variables cannot be used.
- Function references and exponentiation are not allowed.
- Qualification notation is not allowed.

Although the consequent command in an IF can be any valid CID command, it is usually an assignment, PRINT, JUMP, or GO command, as in the following examples:

IF(X.GT.Y+Z)PRINT*,'VALUES ARE',X,Y

Prints the values of X and Y if X is greater than Y+Z.

CYBER INTERACTIVE DEBUG ?set,breakpoint,p.area_l.2 [IN COLLECT MODE ?print*,'input is ',x1,y1,x2,y2,x3,y3 ?pause,'changes?' ?]	Set breakpoint with body at line 2 of AREA.
END COLLECT ?set,trap,store,a [INTERPRET MODE TURNED ON IN COLLECT MODE ?print*,'area is ',a ?]	Set STORE trap with body for variable A.
END COLLECT ?go	Initiete program execution.
INPUT IS 0. 0. 2. 0. 0. 2.	Breakpoint suspends execution; sequence execution initiated.
*B #1, AT P.AREA_L.2 CHANGES?	PAUSE command suspends sequence execution and displays message.
?go	Resume sequence execution.
AREA IS 2.	STORE trap initiates execution of trap body.
INPUT IS 0. 1. .5 2. -1. 1.2 *B #1, AT P.AREA_L.2 CHANGES?	Breakpoint suspends program execution on second pass through AREA; PAUSE command suspends sequence execution.
?x1=1.0 ?y1=2.0	Assign new values to X1 and X1.
?go	Resume sequence execution.
AREA IS .2 INPUT IS 6.1 2. .1 -4. 3.2 7. *B #1, AT P.AREA_L.2 CHANGES? ?x3=4.2 ?go	Third pass through AREA. Assign new velue to X3 and resume execution.
AREA IS 20.7 INPUT IS .2 -2.9 -1.3 8. 5.6 2.8 *B #1, AT P.AREA_L.2 CHANGES? ?go	Fourth pass through AREA. Resume execution. Program runs to completion.
AREA IS 33.705 STOP 20100 MAXIMUM EXECUTION FL. 11.704 CP SECONDS EXECUTION TIME. *T #17, END IN P.RDTR_L.7 ?	

Figure 4-8. Debug Session Illustrating PAUSE Command

```
                    PROGRAM EX          74/74    OPT=0

              1                 PROGRAM EX
              2                 A=1.0
              3                 B=2.0
              4                 CALL SUB(A,B,C)
              5                 D=C**2+1.0
              6                 PRINT*,'VALUES ARE ',A,B,C,D
              7                 END

              1                 SUBROUTINE SUB(X,Y,Z)
              2                 Z=X+Y
              3                 RETURN
              4                 END

CYBER INTERACTIVE DEBUG
?set,breakpoint,l.4

?go

*B #1, AT L.4 <---- Breakpoint suspends execution at line 4 of program EX.
?c=4.0 <----------- Assign value to C.

?go,l.5 <---------- Resume execution at line 5.

VALUES ARE 1. 2. 4. 17.
      END  EX
      15700 MAXIMUM EXECUTION FL.
       .141 CP SECONDS EXECUTION TIME.
*T #17, END IN L.7
?
```

Figure 4-9. Program EX and Debug Session Illustrating GO Command

IF(IFIRST.EQ.1)ZZ=XX*2.0

> If IFIRST is equal to 1, the value XX times 2.0 replaces the current value of ZZ.

IF(A(I).GT.0.0)GO,L.50

> If the value of A(I) is greater than zero, control transfers to line 50 of the program.

IF(A(2).NE.B(2))JUMP,LABL

> If the value of A(2) does not equal the value of B(2), control transfers to the commands following the label LABL in the current command sequence.

Although you can issue an IF command from the terminal, this command is especially powerful when used in command sequences. You can use the IF command to perform a test and conditionally transfer control to another command in the sequence, or to exit from the sequence. The technique for doing this is similar to that of FORTRAN. In FORTRAN, a GO TO statement causes a branch to another executable statement. In CID, the GO or EXECUTE command is used to exit from the sequence, as in the following examples:

IF(A.GT.B)GO

> If the value of A is greater than the value of B, exits from the current sequence and resumes execution of the most recently-suspended process.

IF(I.NE.0)GO,S.20

> If the value of I is not equal to zero, exits from the current sequence and resumes program execution at statement 20.

IF(X+T.LT.Y+S)EXECUTE

> If the value of X+T is less than the value of Y+S, exits from the current sequence and resumes program execution.

The JUMP and LABEL commands are used to transfer control within a sequence.

JUMP AND LABEL COMMANDS

The format of the JUMP command is:

JUMP,name

where name is a label declared in a LABEL command. The function of the JUMP command is identical to the FORTRAN GO TO statement. When CID encounters a JUMP command, control transfers to the command following the label.

A label is established within a command sequence by the following command:

LABEL,name

where name is a string of one through seven letters or digits. The LABEL command is not executed by CID; its sole purpose is to provide a destination for a JUMP command. When a JUMP command is executed, control transfers to the command following the LABEL command.

The JUMP command can be used in conjunction with the IF command to perform a conditional branch, as in the following command sequence example:

```
IF(X.LT.100.0)JUMP,LAB1
X=0.0
GO
LABEL,LAB1
X=X+1.0
```

If the value of X is less than 100.0, 1.0 is added to X and program execution resumes; if X is not less than 100.0, X is set to 0.0 and program execution resumes.

A debug session using the IF, JUMP, and LABEL commands is illustrated in figure 4-10. The program executed to produce this session appears in section 3 (figure 3-4). The purpose of this session is to suspend program execution at the beginning of subroutine SETB and store the value 3.0 into each word of the array B if K is equal to 3. If K is not equal to 3, execution is to proceed normally. To accomplish this, a breakpoint with a body is set at line 3 of SETB. The first command in the body tests K. If K is not equal to 3, program execution resumes at line 3; otherwise, execution of the sequence continues. The remaining commands of the sequence constitute a loop that stores 3.0 into each word of B. The variable K is used as an index and counter since it is not required by the program. When K is equal to the array dimension N, program execution resumes at line 9. A breakpoint is set in the main program at the first subroutine call so that K can be assigned a value of 3.

At this point, you are probably aware that command sequences using the conditional execution capability can become quite complicated. You should, however, attempt to keep sequences short and simple so that you don't spend more time debugging the sequence than would be required to debug your program.

```
CYBER INTERACTIVE DEBUG
?set,breakpoint,p.setb_l.3 [  <---- Set breakpoint at line 3 of SETB and enter collect mode.

IN COLLECT MODE
?if(k.ne.3)go  <---- If K is not equal to 3, resume program execution.

?k=1

?label,aaa  <---- Define label AAA.

?b(k)=3.0

?print*,'b(',k,')=',b(k)

?if(k.ge.n)go,l.10  )
                    )   If all elements of B have been set to 3, resume execution
?k=k+1              )<-- at line 10 of SETB; otherwise, increment counter and make
                    )   another pass through sequence.
?jump,aaa           )

?]

END COLLECT
?set,breakpoint,l.5 )
                    )
?go                 )<-- Suspend execution at line 5 and assign new value to K.
                    )
*B #2, AT L.5       )
?k=3

?go

 B(1 )=3. )
 B(2 )=3. )
 B(3 )=3. )<-- Breakpoint suspends execution at line 3 of PROGY,
 B(4 )=3. )    sequence commands execute.
 B(5 )=3. )
       END  PROGY
       12300 MAXIMUM EXECUTION FL.
       1.968 CP SECONDS EXECUTION TIME.
*T #17, END IN P.PROGY_L.8  <---- Program runs to completion.
?
```

Figure 4-10. Debug Session Illustrating JUMP and LABEL Commands

COMMAND FILES

In addition to executing command sequences established within a debug session, you can execute command sequences stored on a separate file. You can create such a file using a text editor and include any sequence of CID commands in the file. Command files can also be created with the SAVE command (discussed under Saving Trap, Breakpoint, and Group Definitions). There are two reasons why you might want to create a separate file of CID commands:

- By storing commands on a file, you have a permanent copy of the command sequence that can be used for future debug sessions.
- Editing a file of commands using a text editor is easier than editing a sequence of commands in a group or body while executing under CID control. (See Editing a Command Sequence.)

To execute the commands in a file, enter the command:

```
READ,lfn
```

where lfn is the file name. CID reads the file and automatically executes the commands in the same manner as for a group. When execution of the commands is complete, program execution remains suspended, and control returns to you. To resume program execution, enter GO.

Executing commands from a file can be time-consuming since the file must be read each time the command sequence is executed. If a command sequence is to be executed many times in a single session, a more efficient method of executing the commands is to create a command file containing a SET,GROUP command and to include the command sequence in the group. When the file is read by the READ command, the SET,GROUP command is automatically executed and the command sequence is established as a group within the debug session. The group can subsequently be executed without the necessity of reading the file. For example, if a file containing the commands:

```
X1=Y1+Z1
X2=Y2+Z2
PRINT*,X1,X2
```

is created via a text editor and assigned the name COMF, the command READ,COMF must be issued whenever the sequence is to be executed. If, instead, the following file is created:

```
SET,GROUP,GRPX [
X1=Y1+Z1
X2=Y2+Z2
PRINT*,X1,X2
]
```

The command READ,COMF reads the file and causes the SET,GROUP command to be executed, establishing GRPX for the current session. Thereafter, the command READ,GRPX executes the commands in the group and the file COMF is only read once.

The use of text editors under NOS and NOS/BE to create and edit files containing CID commands is described under Editing a Command Sequence.

SAVING TRAP, BREAKPOINT, AND GROUP DEFINITIONS

As with other CID commands, command sequences exist only for the duration of the session in which they are defined. CID provides the capability of saving group, trap, and breakpoint definitions on a separate file. You can print this file or make it permanent. There are two reasons for copying CID definitions to a file:

- To preserve a copy of the definitions for use in the current or in subsequent debug sessions.
- To make it easier to edit command sequences with the system text editor.

The command to save CID definitions has the following forms:

SAVE,BREAKPOINT,lfn,list

> Copies to file lfn the definitions of the breakpoints specified in list; list is an optional list of breakpoint locations (S.n or L.n) or breakpoint numbers (#n) separated by commas. If list is omitted, all breakpoints are saved.

SAVE,TRAP,lfn,type,scope

> Copies to file lfn the definitions of the traps of the specified type defined for the specified scope. Type and scope are optional and are the same as for the SET,TRAP command; if they are omitted, all existing traps are saved.

SAVE,GROUP,lfn,list

> Copies to file lfn the groups specified in list; list is an optional list of group names or numbers (#n) separated by commas. If list is omitted, all groups defined for the current session are saved.

The SAVE command copies the complete definition of the specified traps, breakpoints, or groups to the specified file. (The definition of a trap, group, or breakpoint includes the SET command and any other commands in the body.)

You can combine trap, group, and breakpoint definitions on a single file by specifying the same file name for multiple SAVE commands. A single READ command reestablishes all the definitions stored in the file. Another way to combine definitions on a single file is to enter the command:

SAVE,*,lfn

This command copies all existing trap, group, and breakpoint definitions to the specified file.

Some examples of SAVE commands are as follows:

SAVE,BREAKPOINT,SBPF

> Copies to file SBPF all existing breakpoints.

SAVE,BREAKPOINT,BPFILE,L.10,P.SUBX_S.20

> Copies to BPFILE the definitions of the breakpoints established at line 10 of the home program and statement 20 of subroutine SUBX.

SAVE,BREAKPOINT,FILEA,#2,#5

Copies to FILEA the definition of breakpoints #2 and #5.

SAVE,TRAP,TFILE

Copies to TFILE all existing traps.

SAVE,TRAP,TTT,LINE,P.PROGA

Copies to TTT the definition of the LINE trap established in program unit PROGA.

SAVE,GROUP,GFIL,WRT,RDD,GRPX

Copies to GFIL the definitions of the groups named WRT, RDD, and GRPX.

You can preserve group, trap, and breakpoint definitions across a system logout by making the SAVE file permanent.

Definitions stored on a file can be altered (as described under Editing a Command Sequence) and then restored in the current or in a subsequent session. The command to restore the definitions stored on a file is:

READ,lfn

where lfn is the file containing the definitions. You can issue a READ command in the current session or in a later session. If a READ,lfn is issued in the current session and the definitions previously saved on lfn have not been removed by the appropriate CLEAR command, CID displays a message of the form:

EXISTING BREAKPOINTS WILL BE REDEFINED
OK?

A positive response (YES or OK) causes the existing definitions to be redefined according to the information in the file; a negative response (NO) causes the read command to be ignored.

Note that the READ command only restores the definitions stored in the specified file; it does not cause the commands in the definitions to be executed.

The following READ commands assume that GFIL and TTT are as defined in the preceding examples:

READ,TTT

Restores the LINE trap definition contained in file TTT.

READ,GFIL

Restores the group definitions contained in file GFIL.

A debug session using the SAVE command is illustrated in figure 4-11. The program shown in figure 3-4 in section 3 is executed in debug mode to produce this session. A breakpoint with a body is established in the main program and in subroutine AREA, after which execution is initiated. The program reads the threa records contained in TRFILE. On each pass through the program, the command sequences are executed. After the program terminates, CID gets control because of the END trap, and a SAVE,BREAKPOINT command is issued to save the current breakpoint definitions on the file named AFILE. The session is terminated, the binary file LGO is rewound, and a new session is initiated. The command READ,AFILE restores the breakpoints for the new session. The contents of AFILE are shown in figure 4-12.

The debug sessions in figure 4-13 illustrate the SAVE,GROUP command using the program shown in figure 4-5. The command group PRNT, shown in figure 4-5, is saved on the file named GFILE at the end of the first debug session. At the beginning of the second session, the command READ,GFILE restores the group definition. Breakpoints are set at lines 7, 8, and 9 of MATOP. When each breakpoint is encountered, the command READ,PRNT is issued to execute the group. Note that this command could have been placed in a body for each breakpoint. The groups would then have been executed automatically, without user intervention.

EDITING A COMMAND SEQUENCE

If you wish to make a change to a command sequence in a trap body, breakpoint body, or group, you can remova the definition with the appropriate CLEAR command and reenter the entire sequence. This procedure can be time-consuming for lengthy sequences, however.

CID provides two alternate methods for making changes to a command sequence:

- You can save the trap, breakpoint, or group definition on a separate file and edit the file.
- You can turn on veto mode and edit the sequence interactively. Refer to the CYBER Interactive Debug reference manual for an explanation of this method.

To apply the first method, you must temporarily exit from the current debug session.

SUSPENDING A DEBUG SESSION

CID provides the capability of suspending the current session, returning to system command mode, and resuming the session at a later time. This feature can be used whenever you wish to perform a function outside of CID, but it is especially useful for leaving a session to edit a command sequence.

The commands:

SUSPEND

SUSPEND(lfn)

suspend the current debug session, copy information about the session environment to the specified file, and return control to the operating system. The information includes a copy of the executing program and copies of all opened files, all CID internal tables, and all trap, breakpoint, and group definitions. In short, the file contains all the information necessary to continue the debug session. If lfn is omitted from tha SUSPEND command, the information is written to a local file named ZZZZZDS.

First Session: Breakpoint Definitions Saved.

```
CYBER INTERACTIVE DEBUG
?set,breakpoint,l.5 [

IN COLLECT MODE
?display,#line                          Set breakpoint with body at line 5 of main program.

?list,values,p.rdtr

?]

END COLLECT
?set,breakpoint,l.6                     Set breakpoint at line 6 of RDTR.

?set,breakpoint,p.area_l.7 [

IN COLLECT MODE
?print*,'area is ',a                    Set breakpoint with body at line 7 of subroutine AREA.

?]

END COLLECT
?go                                     Initiate program execution.

#LINE = P.RDTR_L.5                      Breakpoint detected at line 5; sequence execution initiated.
P.RDTR
A = -I,   X1 = 0.0,   X2 = 2.0,   X3 = 0.0,   Y1 = 0.0,   Y2 = 0.0
Y3 = 2.0
 AREA IS 2.
*B #2, AT L.6  (OF P.RDTR)              Breakpoint detected at line 6.
?save,breakpoint,afile                  Copy breakpoint definitions to file AFILE.

?quit

  DEBUG TERMINATED
```

Second Session: Breakpoint Definitions Restored.

```
CYBER INTERACTIVE DEBUG
?read,afile                             Restore breakpoint definitions contained in AFILE.

?list,breakpoint                        List breakpoint locations.

*B #1 = L.5 ,   *B #2 = L.6,   *B #3 = P.AREA_L.7
?go                                     Initiate program execution.

#LINE = P.RDTR_L.5
P.RDTR
A = -I,   X1 = 0.0,   X2 = 2.0,   X3 = 0.0,   Y1 = 0.0,   Y2 = 0.0
Y3 = 2.0
 AREA IS 2.
*B #2, AT L.6  (OF P.RDTR)
?
```

Figure 4-11. Debug Sessions Illustrating SAVE Command

```
SET HOME P.RDTR
SET,BREAKPOINT,L.5 [
DISPLAY,#LINE
LIST,VALUES,P.RDTR
]
SET HOME P.RDTR
SET BREAKPOINT L.6
SET,BREAKPOINT,P.AREA_L.7 [
PRINT*,'AREA IS ',A
]
```

Figure 4-12. Listing of File AFILE

The information contained in the file created by a SUSPEND command is intended for use by CID only; it should not be accessed directly by the user. This file preserves the status of a debug session exactly as it existed when the SUSPEND was executed. The file is a local file; however, you can make the file permanent, thereby preserving the debug session after a logout. The saved debug session can be resumed in a subsequent terminal session. However, except for sessions involving extremely long programs, it should rarely be necessary to continue a debug session over more than one terminal session.

First Session: Group Defined and Saved.

```
CYBER INTERACTIVE DEBUG
?set,group,prnt [          <-- Assign group name and activate collect mode.

IN COLLECT MODE
?message,'contents of mwrk'

?print*,mwrk(1,1),mwrk(1,2),mwrk(1,3)

?print*,mwrk(2,1),mwrk(2,2),mwrk(2,3)      <-- Group body.

?print*,mwrk(3,1),mwrk(3,2),mwrk(3,3)

?]

END COLLECT
?save,group,gfile          <-- Copy group definition to file GFILE.

?quit

  DEBUG TERMINATED
```

Second Session: Group Reestablished.

```
CYBER INTERACTIVE DEBUG
?read,gfile                <-- Restore group definition contained in GFILE.

?list,group                <-- List current group names end numbers.

*G #1 = PRNT
?list,group,#1             <-- List commands in group #1.

*G #1 = PRNT
SET,GROUP,PRNT [
MESSAGE,'CONTENTS OF MWRK'
PRINT*,MWRK(1,1),MWRK(1,2),MWRK(1,3)
PRINT*,MWRK(2,1),MWRK(2,2),MWRK(2,3)
PRINT*,MWRK(3,1),MWRK(3,2),MWRK(3,3)
]
?set,trap,line,l.7...l.8

?go

*T #1, LINE AT L.7
?read,prnt                 <-- Initiate execution of the commands in PRNT.

CONTENTS OF MWRK
 3 3 7
 6 9 5
 4 9 9
?
```

Figure 4-13. Debug Session Illustrating READ and SAVE,GROUP Commands

You should not alter the status of any files used by your program after you issue a SUSPEND command. If you perform any file manipulation operations, such as REWIND, on files used by your program, you might not be able to restart the session normally.

To resume the suspended debug session, enter one of the following commands:

DEBUG(RESUME)

DEBUG(RESUME,lfn)

where lfn is the file name specified in a previous SUSPEND command. If lfn is omitted, CID reads file ZZZZZDS. This command restores the debug session to its status as it existed at the time of suspension. All trap, breakpoint, and group definitions are restored, and all program and debug variables have the values that existed when SUSPEND was executed.

Remember that the most effective debug sessions are short and simple. Thus, it will rarely be necessary to use the SUSPEND/RESUME capability, except to edit command sequences.

EDITING PROCEDURE

To edit a trap body, breakpoint body, or command group, proceed as follows:

1. Save the trap, breakpoint, or group definition with the appropriate SAVE command.
2. Suspend the current session with the SUSPEND command.
3. Use a text editor to make desired changes to the command sequence.
4. Resume the session with the DEBUG(RESUME) command.
5. Remove the old trap, breakpoint, or group definition with the appropriate CLEAR command.
6. Establish the altered definition with the READ command.

After a SUSPEND, be sure that you do not modify or change the position of any files used by your program, because the DEBUG(RESUME) command does not restore these to their status at suspension time.

An example of the procedure for editing a command sequence is shown as performed under NOS/BE (figure 4-14) and NOS (figure 4-15). The purpose of this editing session is to change the command Y=2.0 in the group named AGRP, to Y=3.0. To accomplish this, the group is copied to file SAVFIL, and the debug session is suspended.

Under NOS/BE INTERCOM, the command EDITOR calls the system text editor. The command EDIT,SAVFIL,SEQ makes SAVFIL the edit file and assigns a sequence number to each line in the edit file. The command 140=Y=3.0 makes the desired change to line 140 of the edit file. The edit file is then copied to file NEWFIL and editing is terminated. Refer to the INTERCOM reference manual for more information on the INTERCOM text editor.

Under NOS, the following commands are used to alter GRPFIL:

EDIT,SAVFIL

Enters edit mode to edit SAVFIL.

PRINT*

Lists the edit file.

NEXT 4

Advances the line pointer 4 lines.

REPLACE

Replaces the current line.

PRINT

Lists the current line.

END

Exits from edit mode.

When editing is complete, the debug session is resumed by DEBUG(RESUME) and the group is restored by READ,SAVFIL. Refer to the XEDIT reference manual for detailed information on XEDIT.

INTERRUPTING AN EXECUTING SEQUENCE

The INTERRUPT trap, described in section 3, allows you to gain control at any time during a debug session. If a command sequence is executing at the time of the interrupt, execution of the sequence is suspended and CID displays the message:

INTERRUPTED
?

You can respond as follows:

User Response	CID Action
OK or YES	Resumes sequence execution at the point of the interrupt.
GO or NO,SEQ	Disregards all remaining commands in the sequence and resumes execution of the program.
Any CID command	Executes the specified command and resumes execution of the sequence at the point of the interrupt.

If CID is in the process of displaying information when the interrupt occurs, the information remaining to be printed is lost. A terminal interrupt is therefore an effective means of stopping excessive CID output.

```
CYBER INTERACTIVE DEBUG
?set,group,agrp [

IN COLLECT MODE
?message,'executing group agrp'

?display,#line                          Define group AGRP.

?x=1.0

?y=2.0

?]

END COLLECT
?save,group,savfil,agrp                 Copy definition of AGRP to file SAVFIL.

?suspend                                Suspend debug session.

  DEBUG SUSPENDED


..edit,savfil,seq                       Make SAVFIL the edit file.

..list,all                              List edit file.


   100=SET,GROUP,AGRP [
   110=MESSAGE,'EXECUTING GROUP AGRP'
   120=DISPLAY,#LINE
   130=X=1.0
   140=Y=2.0
   150=]
..140=y=3.0                             Replace line 140.
save,newfil,noseq                       Copy edit file to NEWFIL.

..debug(resume)                         Resume debug session.

CYBER INTERACTIVE DEBUG RESUMED
?clear,group,agrp                       Remove old definition of AGRP.

?read,newfil                            Establish new definition of AGRP by reading NEWFIL.

?list,group,agrp                        List definition of AGRP.

*G #1 = AGRP
SET,GROUP,AGRP [
MESSAGE,'EXECUTING GROUP AGRP'
DISPLAY,#LINE
X=1.0
Y=3.0
]
?
```

Figure 4-14. Editing a Command Sequence Under NOS/BE

```
CYBER INTERACTIVE DEBUG
? set,group,agrp [
 IN COLLECT MODE
? message,'executing group agrp'
? display,#line                          <--- Define group AGRP.
? x=1.0
? y=2.0
? ]
 END COLLECT
? save,group,savfil,agrp <-------------- Copy definition of AGRP to file SAVFIL.
? suspend <----------------------------- Suspend debug session.

SRU       8.979 UNTS.

 RUN COMPLETE.
batch <--------------------------------- Enter batch subsystem.
$RFL,0.
/xedit,savfil <------------------------- Call XEDIT program to edit SAVFIL.
 XEDIT 3.0.9
?? print * <---------------------------- Print edit file.
SET,GROUP,AGRP [
MESSAGE,'EXECUTING GROUP AGRP'
DISPLAY,#LINE
X=1.0
Y=2.0
]
END OF FILE
?? next 4 <----------------------------- Advance line pointer.
Y=2.0
?? replace <---------------------------- Replace current line.
? y=3.0
?? print
Y=3.0
?? end <-------------------------------- Leave edit mode.
SAVFIL  IS A LOCAL FILE
/debug(resume) <------------------------ Resume debug session.
 CYBER INTERACTIVE DEBUG RESUMED
? clear,group,agrp <-------------------- Remove old definition of AGRP.
? read,savfil <------------------------- Establish new definition of AGRP.
? list,group,agrp <--------------------- List definition of AGRP.
 *G #1 = AGRP
 SET,GROUP,AGRP [
 MESSAGE,'EXECUTING GROUP AGRP'
 DISPLAY,#LINE
 X=1.0
 Y=3.0
 ]
?
```

Figure 4-15. Editing a Command Sequence Under NOS

COMMAND SEQUENCE EXAMPLES

Following are two examples of debug sessions that use command sequences. The programs CORR and NEWT, debugged in section 3, are used to illustrate how sequences can be used to speed up the debugging process.

PROGRAM CORR

The original version of CORR, with errors, is shown in figure 3-30 in section 3. Several debug sessions were required to debug the program completely. Commands issued during one session had to be reentered in subsequent sessions. This example demonstrates how this repetition can be eliminated by including the assignment commands in trap and breakpoint bodies and saving the trap and breakpoint definitions on a separate file for use in later sessions. The example also demonstrates how an appropriate command sequence can be used to simulate the reading of input data.

The NOS/BE text editor is used to create the three command files shown in figure 4-16. Each file corresponds to a test case. The files contain assignment commands that insert the correct values for SUMYSQ and N and test values in the arrays X and Y. Two commands, separated by a semicolon, are included on each line. The files are named TEST1, TEST2, and TEST3, respectively.

Listing of TEST1:

```
SUMYSQ=0.0;N=5
X(1)=1.0;Y(1)=1.0
X(2)=10.0;Y(2)=10.0
X(3)=7.6;Y(3)=7.6
X(4)=2.9;Y(4)=2.9
X(5)=5.1;Y(5)=5.1
```

Listing of TEST2:

```
SUMYSQ=0.0;N=5
X(1)=3.0;Y(1)=1.0
X(2)=3.0;Y(2)=5.1
X(3)=3.0;Y(3)=7.6
X(4)=3.0;Y(4)=10.0
X(5)=3.0;Y(5)=15.0
```

Listing of TEST3:

```
SUMYSQ=0.0;N=5
X(1)=0.0;Y(1)=0.0
X(2)=0.0;Y(2)=100.0
X(3)=0.0;Y(3)=0.0
X(4)=0.0;Y(4)=500.0
X(5)=0.1;Y(5)=10.0
```

Figure 4-16. Command Files for Program CORR

Another file named BPFILE, shown in figure 4-17, is created using the text editor. This file contains two breakpoint definitions. The first breakpoint is set at line 15. The PAUSE command will temporarily suspend execution of the breakpoint body. The user can then issue a READ command to execute the commands in TEST1. The command GO,L.22 will resume program execution at line 22, skipping the READ statement. The second breakpoint is set at line 35. The body of this breakpoint contains assignment commands to calculate the correct values for SUMXY and RSQ when the body is executed.

The debug sessions for program CORR are shown in figure 4-18. One session is conducted for each test case. At the beginning of each session, the command READ,BPFILE is issued to establish the breakpoint definitions, and program execution is initiated. When the PAUSE command gives interactive control to the user, a READ command is issued to load the arrays X and Y. Execution is resumed by the GO command, and the commands in the sequences are executed automatically. A LIST,BREAKPOINT is entered in the first session to display existing breakpoints and bodies.

PROGRAM NEWT

The command sequence capability can be applied to the debugging of the Newton's method subroutine shown in figure 3-39 in section 3. Two of the errors in the original program involved an incorrect function name in the subroutine call and an incorrect convergence check that resulted in an infinite loop. The CID commands to correct these errors can be placed in a breakpoint body, as shown in figure 4-19. The sequence includes commands to calculate the correct functional value FX, print the functional value and current number of iterations, test for convergence, and resume program execution at a location following the erroneous statements. If the convergence criterion is satisfied, the PAUSE command will suspend execution of the sequence. The breakpoint, set at line 410, will be encountered on each pass through the loop. Note that although line 410 is illegal because of the unresolved function reference, program execution will be suspended before the statement is executed. The subsequent GO command will resume execution at line 420, bypassing the illegal statement.

After the breakpoint is defined, program execution is initiated with the GO command. The commands in the breakpoint body are executed automatically, as indicated by the PRINT command output, until the convergence criterion is satisfied. In response to the first occurrence of the PAUSE command, GO is entered to resume program execution. In response to the next occurrence of PAUSE, the command EXECUTE,P.MAIN_L.140 is entered to transfer control to line 140 of the main program, allowing the program to print the final results and to terminate.

```
SET,BREAKPOINT,L.15 [
PAUSE,'INPUT?'
PRINT*,'X=',X,' Y=' ,Y
GO,L.22
]
SET,BREAKPOINT,L.35 [
SUMXY=X(1)*Y(1)+X(2)*Y(2)
SUMXY=SUMXY+X(3)*Y(3)+X(4)*Y(4)
SUMXY=SUMXY+X(5)*Y(5)
SUMX=(N*SUMXY-SUMX*SUMY)*(N*SUMXY-SUMX*SUMY)
IF(DENOM.EQ.0.0)PAUSE,'DENOM IS 0'
RSQ=SUMX/DENOM
]
```

Set breakpoint with body at line 15; when PAUSE is executed, user can issue command to read command file.

Set breakpoint with body at line 35; commands are included to calculate correct values for SYMXY and RSQ and to test DEMON for zero value.

Figure 4-17. List of File BPFILE

Debug Session for First Test Case:

```
CYBER INTERACTIVE DEBUG
?read,bpfile                                  Establish breakpoint definitions stored in BPFILE.

?list,breakpoint                              List breakpoint locations.

*B #1 = L.15 ,   *B #2 = L.35
?list,breakpoint,#1,#2                        List breakpoint bodies.

*B #1 = L.15
SET,BREAKPOINT,L.15 [
PAUSE,'INPUT?'
PRINT*,'X=',X,' Y=' ,Y                        Breakpoint #1.
GO,L.22
]
*B #2 = L.35
SET,BREAKPOINT,L.35 [
SUMXY=X(1)*Y(1)+X(2)*Y(2)
SUMXY=SUMXY+X(3)*Y(3)+X(4)*Y(4)
SUMXY=SUMXY+X(5)*Y(5)
SUMX=(N*SUMXY-SUMX*SUMY)*(N*SUMXY-SUMX*SUMY)  Breakpoint #2.
IF(DENOM.EQ.0.0)
PAUSE,'DENOM IS 0'
RSQ=SUMX/DENOM
]
?go

*B #1, AT L.15                                Breakpoint detected at line 15; sequence execution initiated.
INPUT?                                        PAUSE command suspends sequence execution.
?read,test1                                   Initiate execution of commands in TEST1.

?go                                           Resume sequence execution.

 X=1. 10. 7.6 2.9 5.1  Y=1. 10. 7.6 2.9 5.1
*T #17, END IN L.39
?
      END  CORR
      22500 MAXIMUM EXECUTION FL.
      1.598 CP SECONDS EXECUTION TIME.

quit

  DEBUG TERMINATED
```

Debug Session for Second Test Case:

```
CYBER INTERACTIVE DEBUG
?read,bpfile                                  Establish breakpoint definitions stored in BPFILE.

?go

*B #1, AT L.15
INPUT?
?read,test2                                   Initiate execution of commands in TEST2.

?go

 X=3. 3. 3. 3. 3.  Y=1. 5.1 7.6 10. 15.
*B #2, AT L.35
DENOM IS 0
?quit

  DEBUG TERMINATED
```

Figure 4-18. Debug Session Using Command Sequence for Debugging Program CORR (Sheet 1 of 2)

Debug Session for Third Test Case:

```
CYBER INTERACTIVE DEBUG
?read,bpfile <---------------------- Establish breakpoint definitions stored in BPFILE.

?go

*B #1, AT L.15
INPUT?
?read,test3 <----------------------- Initiate execution of commands in TEST3.

?go

 X=0. 0. 0. 0.  .1  Y=0. 100. 0. 500. 10.
*T #17, END IN L.39
?
      END  CORR
      22500 MAXIMUM EXECUTION FL.
      1.689 CP SECONDS EXECUTION TIME.
quit

  DEBUG TERMINATED
```

Figure 4-18. Debug Session Using Command Sequence for Debugging Program CORR (Sheet 2 of 2)

```
 CYBER INTERACTIVE DEBUG
? set,breakpoint,p.newt_l.410 [
 IN COLLECT MODE
? fx=3.0*x-(x+1.0)/(x-1.0)
? print*,'iteration ',its,' fx= ',fx
? if(fx.le..0001)pause,'sequence suspended,check fx'
? if(its.ge.100)pause,'max iterations exceeded'
? go,l.420
? ]
 END COLLECT
? go
  ITERATION 0  FX= 1.
  ITERATION 1  FX= 6.6666666666666E-02
  ITERATION 2  FX= 1.9032332033941E-02
  ITERATION 3  FX= 5.6719259568219E-03
  ITERATION 4  FX= 1.7103829132736E-03
  ITERATION 5  FX= 5.1756717224194E-04
  ITERATION 6  FX= 1.5678125763685E-04
  ITERATION 7  FX= 4.7507133633218E-05
 *B #1, AT P.NEWT_L.410
 SEQUENCE SUSPENDED,CHECK FX
? go
  ITERATION 8  FX= 1.4396770417591E-05
 *B #1, AT P.NEWT_L.410
 SEQUENCE SUSPENDED,CHECK FX
? execute,p.main_l.140
 CONVERGENCE IN    8 ITERATIONS.  X=   -.2152E+00
 *T #17, END IN P.MAIN_L.200
? quit

SRU     28.370 UNTS.

 RUN COMPLETE.
```

Set breakpoint with body at line 410. Include commands to calculate FX, test for convergence, test number of iterations, and resume execution at line 420.

Initiate execution.

The breakpoint body is executed on each pass through the loop, until the IF test is satisfied and the PAUSE command suspends execution of the sequence.

Resume execution of main program. In this case, EXECUTE must be used. A GO command entered here would resume execution of the suspended sequence.

Figure 4-19. Debug Session Using Command Sequence for Debugging Subroutine NEWT

DEBUGGING IN AN OVERLAY ENVIRONMENT 5

Programs containing overlays can be executed under CYBER Interactive Debug (CID) control using all the features presented in the preceding sections. In addition, CID provides the following features to facilitate debugging of overlays:

- Qualification notation which allows you to reference locations in different overlays.
- An OVERLAY trap that suspends program execution when an overlay is loaded.
- Special command forms that limit the command scope to specific overlays.

An important fact to remember when debugging programs containing overlays is that while all CID commands are valid for overlays in memory, only certain commands can reference locations in overlays that are not loaded.

SUMMARY OF OVERLAY PROCESSING

Overlaying allows you to divide a program into sections, called overlays, to reduce the amount of memory required for execution. Different overlays can occupy the same storage locations at different times. Thus, when an overlay residing in memory is not currently required by the program, it can be replaced by another overlay.

There are three levels of overlays: a zero level, a primary level, and a secondary level. The zero level, sometimes referred to as the main overlay, is resident in memory throughout program execution. The primary level is called from the zero level and is loaded immediately above the zero level. The secondary level is called from its associated primary level or from the zero level and is loaded immediately above the primary level.

A primary level overlay can have up to 63 secondary level overlays associated with it. When a primary overlay is called from the zero level overlay, it replaces the primary overlay currently residing in memory. When a secondary level overlay is called from a zero level or primary level, it replaces the secondary overlay currently residing in memory. Thus, only the zero level, one primary level, and one secondary level can reside in memory concurrently.

Overlays are identified by a pair of integers, as follows:

(0,0)	Zero or main overlay
(i,0)	Primary overlay
(i,j)	Secondary overlay

where i is the primary level number and j is the secondary level number. For example, (1,0) and (2,0) are primary overlays; (2,1), (2,2), and (2,3) are secondary overlays associated with primary overlay (2,0).

A group of program units to be loaded into an overlay must be preceded by an OVERLAY directive of the form:

OVERLAY(lfn,i,j)

where lfn is the name of the file on which the overlay is to be written, and i and j are level numbers. The OVERLAY directive must begin in column 7.

Overlays are called from within a FORTRAN program by the statement:

CALL OVERLAY(lfn,i,j)

where lfn is the name of the file in character string format on which the overlay is written, and i and j are level numbers.

An example of a program containing overlays is illustrated in figure 5-1. This program contains a main overlay (0,0), two primary level overlays (1,0) and (2,0), and a secondary overlay (1,1) associated with overlay (1,0). The overlays are stored on a file named OVLF, as established by the OVERLAY directives. Each overlay contains a program unit that performs a simple computation. The variables X, Y, and Z are declared in common and are therefore global to the program. The variable RESULT is referenced in three program units and is local to each. The program units in both primary overlays have the same name.

```
OVERLAY(OVLF,0,0)
PROGRAM SETXYZ
COMMON /ACOM/X,Y,Z
X = 1.0
Y = 2.0
Z = 3.0
CALL OVERLAY('OVLF',1,0)
CALL OVERLAY('OVLF',2,0)
END
C
OVERLAY(OVLF,1,0)
PROGRAM COMP
COMMON /ACOM/X,Y,Z
RESULT = -3.0*X - 2.0*Y + 2.0*Z
CALL OVERLAY('OVLF',1,1)
RETURN
END
C
OVERLAY(OVLF,1,1)
PROGRAM COMP2
COMMON /ACOM/ R,S,T
RESULT = 5.0*R - 6.0*S + 4.0*T
RETURN
END
C
OVERLAY(OVLF,2,0)
PROGRAM COMP
COMMON /ACOM/A,B,C
RESULT = 4.0*A + 2.0*B - C
RETURN
END
```

Figure 5-1. Sample Overlay Program

The program calculates a value for the variable RESULT in each overlay. Each calculation is local to the overlay in which it resides. The (0,0) overlay is loaded first and remains in memory throughout execution. This overlay sets values for X, Y, and Z and then calls the (1,0) overlay. The (1,0) overlay calculates RESULT, then calls the (1,1) overlay which calculates its local RESULT. At this point, the three overlay levels reside in memory concurrently. When execution of the (1,1) overlay has completed, control returns to the main overlay which then calls the second primary overlay (2,0). Overlay (2,0) replaces overlays (1,0) and (1,1) in memory.

Refer to the CYBER Loader reference manual or to the FORTRAN reference manual for more information on overlays.

QUALIFICATION

The notation forms presented in section 3 (table 3-1) are valid for programs containing overlays. However, when referencing a location in a program unit having tha same name as a program unit in a different overlay, you must indicate the overlay containing the desired program unit. This is accomplished by prefixing the location specification with an overlay qualifier of the form:

(i,j)

where i and j are the overlay level numbers.

Examples:

SET,TRAP,LINE,(2,0)P.MTADD

Sets a LINE trap in program unit MTADD residing in overlay (2,0).

SET,BREAKPOINT,(2,1)P.MTADD_L.5

Sets a breakpoint at line 5 of program unit MTADD residing in overlay (2,1).

DISPLAY,(0,0)P.SUB1_X

Displays the contents of X in program unit SUB1 residing in the zero level overlay.

The following restrictions apply to the use of overlay qualification notation:

- It is necessary to use overlay qualification only when duplicate program unit names exist.
- Overlay qualification cannot be used with the PRINT, IF, and assignment commands.
- If the overlay qualifier is omitted from a location specification and duplicate program unit names exist, the name of the program unit currently in memory is selected. If the named program unit is not in memory, the program unit residing in the overlay having the lowest primary level number is selected.

Overlay qualification notation is also used in CID output messages to denote a particular overlay, as in the trap message:

*T #17, END IN (0,0)P.SETXY_L.9

An END trap has occurred at line 9 of program SETXY in overlay (0,0).

REFERENCING LOCATIONS IN UNLOADED OVERLAYS

The following restrictions apply to overlays that have not been loaded into memory:

- You can set traps and breakpoints in any overlay, even if it is not in memory. These traps and breakpoints will be recognized when the containing overlay is loaded and its programs are executed.
- You cannot resume execution in an unloaded overlay.
- You cannot display the contents of locations within overlays that are not in memory. The LIST,VALUES command lists only those variables defined in loaded overlays. LIST,VALUES displays variables in alphabetical order, grouped according to the program unit in which they are defined. Each program unit name is prefixed by an overlay qualifier of the form (i,j).
- You cannot alter the contents of variables within overlays that are not in memory.

The following commands can reference locations in unloaded overlays:

- SET,TRAP
- SET,BREAKPOINT
- SET,HOME
- LIST,TRAP
- LIST,BREAKPOINT
- LIST,MAP
- CLEAR,TRAP
- CLEAR,BREAKPOINT
- SAVE,TRAP
- SAVE,BREAKPOINT

If you illegally reference a location in an unloaded overlay, CID issues the error message:

*ERROR - ADDRESS IN UNLOADED OVERLAY

OVERLAY TRAP

The OVERLAY trap suspends program execution and gives control to CID whenever specified overlays are loaded into memory. This allows you to examine and alter the status of a program as it exists at the time the overlay is loaded. The trap occurs after the overlay is loaded but before control transfers to the loaded overlay. The forms of the command to set an overlay trap are:

SET,TRAP,OVERLAY,*

CID gets control when any overlay is loaded.

SET,TRAP,OVERLAY,(i,j)

CID gets control when overlay (i,j) is loaded.

When an overlay trap occurs, CID issues the message:

T #n, OVERLAY (i,j) IN (i,j)P.name_L.0

n	Trap number assigned by CID.
i,j	Level numbers of the current overlay.
name	Name of the program unit in the current overlay to be executed first.

COMMAND FORMS FOR OVERLAY DEBUGGING

The commands to LIST, CLEAR, and SAVE traps and breakpoints, and the LIST,MAP command have special forms intended for use with overlay programs. These forms, listed in table 5-1, allow you to specify an overlay or list of overlays for the scope parameter.

The LIST,MAP command is especially useful when used with overlay programs. A special form of this command lists all program modules and groups them according to overlay. Overlays are identified by an (i,j) designation. Overlays currently in memory are indicated by an asterisk.

OVERLAY EXAMPLE

A debug session for the program in figure 5-1 is illustrated in figure 5-2. The purpose of this session is to suspend program execution after each overlay is loaded and to issue various commands to examine the status of the program. Control statements are included to activate debug mode and compile, load, and execute the program. Debug mode must be on at compile time and at the time the load sequence is issued.

The following CID commands are issued during this session:

LIST,MAP

Lists the level numbers of all overlays in the program. Overlays currently in memory are indicated by an asterisk.

LIST,MAP,(2,0)

Lists program modules contained in the (2,0) overlay.

SET,TRAP,OVERLAY,*

Sets an overlay trap that suspends execution when any overlay is loaded.

TRACEBACK

Displays a program traceback list starting at the home program.

DISPLAY,#HOME

Displays the name of the home program and the overlay in which it resides.

SET,HOME,(1,0)P.COMP

Designates COMP in overlay (1,0) as the home program. Note that the (1,0) overlay is not in memory when this command is issued. Variables declared in COMP cannot be referenced, as illustrated by the next command.

PRINT*,RESULT

Attempts to print the value of RESULT. The attempt fails because the overlay containing the current home program is not in memory.

TABLE 5-1. COMMAND FORMS FOR OVERLAY PROGRAMS

Command	Description
LIST,TRAP,type,(i,j),(i,j),...	Lists addresses of traps in the specified overlays; * can be substituted for type, in which case all types are listed.
LIST,MAP,(i,j),(i,j),...	Lists program modules contained in the specified overlays.
LIST,BREAKPOINT,(i,j),(i,j),...	Lists locations of all breakpoints in the specified overlays.
CLEAR,TRAP,type,(i,j),(i,j),...	Clears all traps of the specified type in the specified overlays.
CLEAR,BREAKPOINT,(i,j),(i,j),...	Clears all breakpoints in the specified overlays.
SAVE,BREAKPOINT,lfn,(i,j),(i,j),...	Copies the breakpoints in the specified overlays to lfn.
SAVE,TRAP,lfn,type,(i,j),(i,j),...	Copies the traps of the specified type in the specified overlays to lfn; * can be substituted for type, in which case all types are copied.

```
 debug
$DEBUG.
/ftn5(i=ovprog,l=listf,seq)
     0.303 CP SECONDS COMPILATION TIME.
/enter_load(lgo) lgo <------------------------------ Load sequence.
 CYBER INTERACTIVE DEBUG
? list,map
 (0,0) * ,    (1,0),    (1,1),    (2,0) <---------- Overlay (0,0) is in memory.
? set,trap,overlay,* <----------------------------- Set OVERLAY trap.
? go
 *T #1, OVERLAY (1,0) IN (1,0)P.COMP_L.0 <--------- OVERLAY trap occurs when overlay (1,0) is loaded.
? list,map
 (0,0) * ,    (1,0) * ,    (1,1),    (2,0) <------ Overlays (0,0) and (1,0) are in memory.
? go
 *T #1, OVERLAY (1,1) IN (1,1)P.COMP2_L.0 <-------- OVERLAY trap occurs when overlay (1,1) is loaded.
? list,map
 (0,0) * ,    (1,0) * ,    (1,1) * ,    (2,0) <--- Overlays (0,0), (1,0), and (1,1) are in memory.
? display,#home
 #HOME = (1,1)P.COMP2 <---------------------------- Program COMP2 in overlay (1,1) is current home program.
? list,values <------------------------------------ List all variables and values currently in memory.
 (0,0)P.SETXYZ
 X = 1.0,    Y = 2.0,    Z = 3.0
 (1,0)P.COMP
 RESULT = -1.0,    X = 1.0,    Y = 2.0,    Z = 3.0
 (1,1)P.COMP2
 R = 1.0,    RESULT = -1,    S = 2.0,    T = 3.0
? traceback <-------------------------------------- Initiate traceback from home program.
 P.COMP2 CALLED FROM P.COMP_L.240
 P.COMP CALLED FROM P.SETXYZ_L.160
? go
 *T #1, OVERLAY (2,0) IN (2,0)P.COMP_L.0 <--------- OVERLAY trap occurs when overlay (2,0) is loaded.
? list,map
 (0,0) * ,    (1,0),    (1,1),    (2,0) * <-------- Overlays (0,0) and (2,0) in memory.
? go
 *T #17, END IN (0,0)P.SETXYZ_L.180 <-------------- Program terminates at line 180 of program SETXYZ in overlay (0,0).
? list,map
 (0,0) * ,    (1,0),    (1,1),    (2,0) *
? list,values <------------------------------------ List all variables and values currently in memory.
 (0,0)P.SETXYZ
 X = 1.0,    Y = 2.0,    Z = 3.0
 (2,0)P.COMP
 A = 1.0,    B = 2.0,    C = 3.0,    RESULT = 5.0
? print*,result
 *ERROR - NO PROGRAM VARIABLE RESULT <------------- Variable RESULT is not defined in home program.
? set,home,(1,0)p.comp <--------------------------- Make program COMP in overlay (1,0) the home program.


? print*,result
 *ERROR - ADDRESS IN UNLOADED OVERLAY <------------ Home program is not in memory.
? set,home,(2,0)p.comp <--------------------------- Make program COMP in overlay (2,0) the home program.
? print*,result
  5.
? quit
 DEBUG TERMINATED
```

Figure 5-2. Debug Session for Overlay Program

STANDARD CHARACTER SETS A

Control Data operating systems offer the following variations of a basic character set:

- CDC 64-character set
- CDC 63-character set
- ASCII 64-character set
- ASCII 63-character set

The set in use at a particular installation is specified when the operating system is installed.

Depending on another installation option, the system assumes an input deck has been punched either in 026 or in 029 mode (regardless of the character set in use). Under NOS/BE, the alternate mode can be specified by a 26 or 29 punched in columns 79 and 80 of the job statement or any 7/8/9 card. The specified mode remains in effect throughout the job unless it is reset by specification of the alternate mode on a subsequent 7/8/9 card.

Under NOS, the alternate mode can be specified by a 26 or 29 punched in columns 79 and 80 of any 6/7/9 card, as described above for a 7/8/9 card. In addition, 026 mode can be specified by a card with 5/7/9 multipunched in column 1; 029 mode can be specified by a card with 5/7/9 multipunched in column 1 and a 9 punched in column 2.

Graphic character representation appearing at a terminal or printer depends on the installation character set and the terminal type. Characters shown in the CDC Graphic column of table A-1 are applicable to BCD terminals; ASCII graphic characters are applicable to ASCII-CRT and ASCII-TTY terminals.

TABLE A-1. FORTRAN AND STANDARD CHARACTER SETS

FORTRAN	Display Code (octal)	CDC Graphic	CDC Hollerith Punch (026)	CDC External BCD Code	ASCII Graphic Subset	ASCII Punch (029)	ASCII Code (octal)
: (colon)	00†	: (colon) ††	B-2	00	: (colon) ††	8-2	072
A	01	A	12-1	61	A	12-1	101
B	02	B	12-2	62	B	12-2	102
C	03	C	12-3	63	C	12-3	103
D	04	D	12-4	64	D	12-4	104
E	05	E	12-5	65	E	12-5	105
F	06	F	12-6	66	F	12-6	106
G	07	G	12-7	67	G	12-7	107
H	10	H	12-8	70	H	12-8	110
I	11	I	12-9	71	I	12-9	111
J	12	J	11-1	41	J	11-1	112
K	13	K	11-2	42	K	11-2	113
L	14	L	11-3	43	L	11-3	114
M	15	M	11-4	44	M	11-4	115
N	16	N	11-5	45	N	11-5	116
O	17	O	11-6	46	O	11-6	117
P	20	P	11-7	47	P	11-7	120
Q	21	Q	11-8	50	Q	11-8	121
R	22	R	11-9	51	R	11-9	122
S	23	S	0-2	22	S	0-2	123
T	24	T	0-3	23	T	0-3	124
U	25	U	0-4	24	U	0-4	125
V	26	V	0-5	25	V	0-5	126
W	27	W	0-6	26	W	0-6	127
X	30	X	0-7	27	X	0-7	130
Y	31	Y	0-8	30	Y	0-8	131
Z	32	Z	0-9	31	Z	0-9	132
0	33	0	0	12	0	0	060
1	34	1	1	01	1	1	061
2	35	2	2	02	2	2	062
3	36	3	3	03	3	3	063
4	37	4	4	04	4	4	064
5	40	5	5	05	5	5	065
6	41	6	6	06	6	6	066
7	42	7	7	07	7	7	067
8	43	8	B	10	8	B	070
9	44	9	9	11	9	9	071
+ (plus)	45	+	12	60	+	12-8-6	053
- (minus)	46	-	11	40	-	11	055
* (asterisk)	47	*	11-B-4	54	*	11-8-4	052
/ (slash)	50	/	0-1	21	/	0-1	057
((left paren)	51	(	0-8-4	34	(	12-8-5	050
) (right paren)	52	)	12-8-4	74	)	11-8-5	051
$ (currency)	53	$	11-8-3	53	$	11-8-3	044
= (equals)	54	=	B-3	13	=	8-6	075
blank	55	blank	no punch	20	blank	no punch	040
, (comma)	56	, (comma)	0-8-3	33	, (comma)	0-8-3	054
. (decimal point)	57	. (period)	12-8-3	73	. (period)	12-8-3	056
	60	≡	0-8-6	36	#	B-3	043
	61	[	B-7	17	[	12-8-2	133
	62	]	0-8-2	32	]	11-8-2	135
	63	% ††	8-6	16	% ††	0-8-4	045
" (quote)	64	≠	8-4	14	" (quote)	B-7	042
	65	↱	0-B-5	35	_ (underline)	0-8-5	137
	66	∨	11-0 or 11-8-2†††	52	!	12-8-7 or 11-0†††	041
	67	∧	0-B-7	37	&	12	046
' (apostrophe)	70	↑	11-8-5	55	' (apostrophe)	8-5	047
	71	↓	11-8-6	56	?	0-8-7	077
	72	<	12-0 or 12-8-2†††	72	<	12-8-4 or 12-0†††	074
	73	>	11-8-7	57	>	0-8-6	076
	74	≤	8-5	15	@	B-4	100
	75	≥	12-8-5	75	\	0-8-2	134
	76	¬	12-8-6	76	~ (circumflex)	11-8-7	136
	77	; (semicolon)	12-8-7	77	; (semicolon)	11-8-6	073

†Twelve zero bits at the end of a 60-bit word in a zero byte record are an end-of-record mark rather than two colons.
††In installations using a 63-graphic set, display code 00_B has no associated graphic or card code; display code 63_B is the colon (8-2 punch). The % graphic and related card codes do not exist and translations yield a blank (55_B).
†††The alternate Hollerith (026) and ASCII (029) punches are accepted for input only (NOS/BE only).

GLOSSARY

Abort -
To terminate a program or job when an error condition (hardware or software) exists from which the program or computer cannot recover.

Auxiliary File -
An optional file, established by the SET,AUXILIARY command, to which CYBER Interactive Debug (CID) output is written. The output types written to this file are specified by special output codes.

Batch Mode -
A mode of CID execution which allows programs intended for batch execution to be executed under CID control.

Breakpoint -
A designated location in a program where execution is to be suspended.

Collect Mode -
A mode of CID execution in which commands entered by the user are not executed, but are included in a group, trap, or breakpoint body. Collect mode is initiated by a left bracket ([) at the end of a SET,TRAP; SET,GROUP; or SET,BREAKPOINT command, and are terminated by a right bracket (]).

Common Block -
A module intended solely for storing data. A block of data can be declared in common to both the calling routine and the called routine as an alternative to passing data to routines via parameter values.

Debug Mode -
A mode of execution in which special CID tables are generated during compilation and in which user programs are executed under CID control; initiated by a DEBUG(ON) control statement, and terminated by a DEBUG(OFF) control statement.

Debug Session -
A sequence of interactions between the user and CID, beginning when execution of the user program is initiated in debug mode, and ending when a QUIT command is issued.

Entry Point -
A special location within a program module. In FORTRAN programs, this location is named in an ENTRY statement, FUNCTION statement, or SUBROUTINE statement. An entry point is, by convention, the target of the RJ (Return Jump) instruction that transfers control to the module. The RJ instruction stores the return address at the entry point location and starts execution at the next location.

Extended Memory -
An additional memory available as an option on CYBER computer systems. This memory can only be used for program and data storage, not for program execution. Special hardware instructions exist for transferring data between central memory and extended memory.

Group -
A sequence of CID commands established and assigned a name by a SET,GROUP command and executed when a READ command is issued.

Home Program -
Program unit in which variables, line numbers, and statement labels referenced by the user in CID commands are assumed to be located unless appropriate qualifiers appear. By default, the home program is the program unit being executed at the time CID gains control. The user can change the default with the SET,HOME command.

Interactive -
Capable of a two-way back and forth exchange of information.

Interactive Mode -
The normal mode of CID execution. The user enters commands directly from the terminal and CID immediately executes the commands. CID can also execute in batch mode.

Interpret Mode -
A mode of execution in which a special routine, called an interpreter, examines each machine instruction to be executed in the user program, and simulates its execution by the execution of several of its own instructions. Execution in interpret mode consequently takes 20 to 50 times as long as direct execution. Certain CID features require interpret mode execution.

Interrupt (verb) -
To stop a running program in such a way that it can be resumed at a later time.

Interrupt (noun) -
A special control signal which, when issued, causes action as described for INTERRUPT (verb).

Module -
A named section of coding output by a compiler or assembler, or a block of data (common block). Prior to loading, modules are called object modules; after loading they are called load modules. In FORTRAN, a module is a main program, subroutine, function subprogram, or common block.

Optimizing Mode -
One of the compilation modes of the FORTRAN compiler, indicated by the control statement options OPT=0, 1, 2, or 3. OPT=0 compilation is required for full use of the features described in this manual.

Overlay -
A portion of a program, consisting of one or more modules, which can share an allocated area of memory with other portions of the program. When access to a particular module is required, the overlay containing that module is loaded, thus overlaying the previous contents of the memory area allocated for that overlay. Such a scheme allows large programs to execute in a limited amount of memory.

Program -
The completely loaded set of one or more object modules. A program that has been loaded in debug mode can be executed under CID control.

Program Module -
A module intended for execution. A program module always has an entry point, a named location in the module to be used in calling the module.

Program Unit -
A FORTRAN main program, function subprogram, subroutine, or block data subprogram.

Terminal Session -
The sequence of interactions between the user and a terminal which begins when the user logs in, and terminates when the user logs out. Contrast with Debug Session.

Trap (verb) -
To suspend program execution and transfer control to CID upon the detection of a specified condition.

Trap (noun) -
A mechanism that detects the occurrence of a specified condition, suspends execution of the user program at that point, and transfers control to CID.

The following paragraphs discuss some of the errors that can cause a FORTRAN program to terminate prematurely. To use CID effectively, you should be able to recognize situations that can cause these errors. A knowledge of the internal representation of numbers can be helpful in understanding why arithmetic errors occur.

FLOATING-POINT REPRESENTATION

The internal floating-point format is shown in figure C-1. Bits 0 through 47 contain the coefficient of the number, equivalent to about 14 decimal digits. The binary point is assumed to be at the right of bit 0. The sign of the coefficient is represented by bit 59, 0 for a positive value and 1 for a negative value. The exponent is contained in bits 48 through 58 and is biased by 2000 octal, that is, 2000 octal is added to the exponent. Some examples of internal floating-point representation, including the largest and smallest permissible values, are illustrated in table C-1. Operands exceeding the maximum or minimum values cause execution errors.

ARITHMETIC MODE ERRORS

Arithmetic mode errors occur when the central processor encounters an instruction that cannot be executed. Such an instruction generally involves an operand that contains invalid data or an address beyond the user's field length. The arithmetic mode errors and possible causes are listed in table C-2.

When a mode error occurs, the executing program is aborted and a message of the following format is issued:

time ERROR MODE=n ADDRESS=xxxxxx

where n is the mode number and xxxxxx is the relative octal address where the error occurred.

When a mode error occurs while executing in debug mode, control passes to CID. This is because of the ABORT trap explained in section 3. Upon receiving control, CID issues the following message and prompt:

*T#18 ABORT CPU ERROR EXIT n IN L.m
?

where n is the mode number and m is the source line number where the error occurred. You can then enter CID commands to examine the status of the program as it existed at the time of termination.

OVERFLOW, UNDERFLOW, DIVISION BY ZERO

Floating-point overflow occurs when a value is generated which exceeds the allowable exponent range. This situation can occur when performing calculations with numbers of extremely large magnitude or when a nonzero number is divided by zero.

When overflow occurs, the exponent is set to 3777_8 (the largest possible exponent) and the characteristic is set to zero. Such an operand is called an infinite operand. The executing program aborts when the infinite operand is used in a subsequent computation, not when it is generated. A debug session for a program that generates an infinite operand is illustrated in figure C-2. A division by zero generates the infinite operand. Program execution terminates when the infinite operand is referenced in the statement D=C+1.0. The LIST,VALUES command shows the program variables as they existed at the time of termination. The value of the variable C, the infinite operand, is represented by the letter R.

Floating-point underflow occurs when a value is generated which would have an exponent less than -293. The resulting operand is set to all zeros.

The allowable range for floating-point numbers is shown in table C-3.

TABLE C-1. EXAMPLES OF FLOATING-POINT NUMBERS

Number	Internal Representation
+1.	1720 4000 0000 0000 0000
+100.	1726 6200 0000 0000 0000
-100.	6051 1577 7777 7777 7777
1.E64	2245 6047 4037 2237 7733
-1.E64	6404 2570 0025 6605 5317
0.	0000 0000 0000 0000 0000

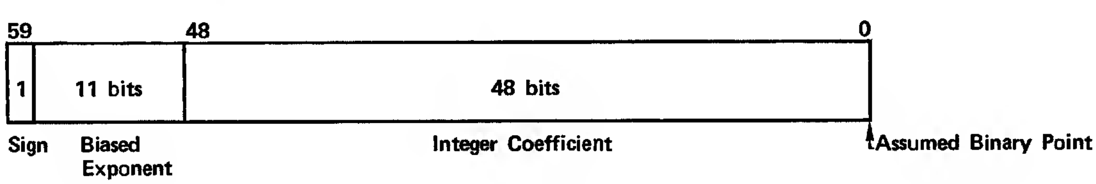

Figure C-1. Internal Floating-Point Format

TABLE C-2. MODE ERRORS

Error Number	Dayfile Message (NOS)	Explanation	Possible Causes
0	ILLEGAL INSTRUCTION	A branch to location zero has occurred or an illegal instruction has been executed.	Machine instructions destroyed by array overflow; mismatch between number of arguments in CALL and SUBROUTINE statement.
1	CM OUT OF RANGE	Program has referenced a location outside the user's field length.	a. Program attempted to fetch or store data outside program field length; probably caused by incorrect subscript. b. Program attempted to jump to an address outside program field length; probably caused by missing function or subroutine or misspelled function or subroutine name. c. Program executed a word that did not contain an instruction; probably caused by array overflow.
2	ARITHMETIC OVERFLOW	Infinite value used as operand.	Nonzero number divided by zero; very large number divided by very small number; very large number multiplied by very large number; numbers very near 10^{322} in absolute value added or subtracted; large number raised to a large power; at some installations, undefined value used in a calculation.
4	ARITHMETIC INDEFINITE	Indefinite value used as operand.	Zero divided by zero; zero multiplied by an infinite value; infinite value divided by an infinite value; infinite values added or subtracted; at some installations, undefined value used in a calculation.
3		Combination of 1 and 2.	See above.
5		Combination of 1 and 4.	See above.
6		Combination of 2 and 4.	See above.
7		Combination of 1, 2, and 4.	See above.

INDEFINITE OPERANDS

Indefinite operands are generated when the central processor encounters an instruction that cannot be resolved, such as a division where both the dividend and the divisor have a value of zero. Indefinite operands can also be generated when a variable has not been initialized (the value assigned to uninitialized areas of memory is an installation parameter). As with infinite operands, indefinites cause abnormal termination of execution when they are referenced.

An indefinite operand is represented by the character I when displayed. The internal representation of an indefinite operand is shown in table C-3.

ERRORS INVOLVING INTEGERS

The maximum permissible absolute value of an integer depends on the context in which it is used. When an integer exceeds the limits of the central processor, it is assigned a value of zero. The allowable range for integers is shown in table C-4.

```
                    PROGRAM ERR          74/74    OPT=0

              1                 PROGRAM ERR
              2                 A=1.0
              3                 B=1.0
              4                 C=(A+B)/(A-B)
              5                 D=C+1.0
              6                 END



         CYBER INTERACTIVE DEBUG
         ?go

         *T #18, ABORT CPU ERROR EXIT 02 IN L.5 <--- Error exit occurs in line 5.
         ?list,values

         P.ERR
         A = 1.0,    B = 1.0,    C = R,    D = R
         ?
```

Figure C-2. Program and Debug Session Illustrating Mode Error

TABLE C-3. FLOATING-POINT REPRESENTATIONS

Value	Positive Operand		Negative Operand	
	Floating-Point Representation	Octal	Floating	Octal
Largest Value	1.265014083171E+322	$37767...7_8$	-1.265014083171E+322	$40010...0_8$
Smallest Value	3.131513062514E-293	$000140...0_8$	-3.131513062514E-294	$777637...7_8$
Zero (underflow yields zero operand)	0.0	$0...0_8$	-0.0	$7...7_8$
Overflow (infinite operand)	R	$37770...0_8$	-R	$4000...0_8$
Indefinite	I	$17770...0_8$	-I	$60007...7_8$

TABLE C-4. INTEGER REPRESENTATIONS

Value	Positive Operand		Negative Operand	
	Integer	Octal	Integer	Octal
Maximum value for arithmetic operations	$2^{48}-1$ (5761 7927 7326 7128 31)	$37767...7_8$	$-(2^{48}-1)$	$40010...0_8$
Maximum value for subscripts and DO loop index	$2^{17}-1$ (131071)	$0...037777_8$	Negative operand not permitted	
Zero	0	0...0	-0	$7...7_8$
Infinite Operand		Set to zero		
Indefinite Operand		Set to zero		

BATCH MODE DEBUGGING D

CYBER Interactive Debug (CID) is primarily intended to be used interactively, but can be used in batch mode. Possible reasons for using batch mode include the possibility of a large volume of output, or lack of access to a terminal. In batch mode, however, you must plan the entire session in advance. This requires care and a knowledge of what errors are likely to occur.

To conduct a debug session in batch mode, commands must exist on a file of card images called DBUGIN from which CID reads all input. You can create this file by using the system text editor, or you can punch the commands on cards, include them as part of the job deck, and copy file INPUT to DBUGIN. Commands are punched or written in the same format as in interactive mode; a card can contain a single command or multiple commands separated by semicolons.

As in interactive execution, debug mode is established by the DEBUG control statement. The debug session is initiated by a statement to load and execute the program. Control transfers immediately to CID, which begins executing the commands in DBUGIN. When CID encounters a GO or EXECUTE in the command stream, control transfers to the user program. The user program executes until a trap or breakpoint is encountered. In this manner, control transfers between the program and CID with no user intervention.

A QUIT command is normally the last command of the sequence. However, this command can be omitted and CID will terminate after the last command has been executed.

Following are some restrictions that apply to batch mode debugging:

- Invalid commands are disregarded; when CID encounters such a command, processing continues with the next command.
- Commands that would generate a warning message in interactive mode are executed in batch mode.
- All commands are executed except when execution is impossible; you cannot establish veto mode in a batch session.

In batch mode, all output from CID is written to a file named DBUGOUT. This is a local file and it is the user's responsibility to print the file or make it permanent. You can control the types of output sent to DBUGOUT with the SET,OUTPUT command. Output can also be sent to a separate file with the SET,AUXILIARY command.

Batch output from a debug session does not normally show the user-specified CID commands as they are executed. CID reads the commands from DBUGIN but does not echo them to DBUGOUT unless the T option is specified on the SET,OUTPUT command. Use of this option usually improves the readability of a batch debug session.

With the exception of the SET,VETO command, all CID commands are valid in batch mode. You can set traps and breakpoints, define command sequences, display and alter the values of program variables, and resume program execution. The commands in DBUGIN should be specified in the same order as in interactive mode. CID accesses DBUGIN for all input that would normally be input from the terminal.

A suggested technique for batch mode debugging is to use only traps and breakpoints with bodies. This way, the commands to be executed on suspension of execution appear in the input stream immediately after the SET,TRAP or SET,BREAKPOINT command that caused suspension. In addition, only one GO command is required.

An example of a program to be debugged in batch mode (under NOS) is illustrated in figure D-1. (To execute this program under NOS/BE, replace the job user, and change statements with a job statement containing the appropriate accounting information.) Breakpoints with bodies are set initially at lines 2 and 5, and program execution is initiated. When the first breakpoint is encountered, CID receives control, executes commands in the body, and returns control to the program. The command GO,L.4 skips the FORTRAN statements that open and read an input file. When the breakpoint at line 5 is encountered, CID executes the LIST,VALUES and QUIT commands. The contents of the output file DBUGOUT are shown in figure D-2.

```
JOB Statement
USER Statement
CHARGE Statement
COPYBR(INPUT,DBUGIN)
REWIND(DBUGIN)
DEBUG(ON)
FTN5.
LGO.
REWIND(DBUGOUT)
COPYSBF(DBUGOUT,OUTPUT)
7/8/9 in column 1
SET,BREAKPOINT,L.2 (
X1=0.0;Y1=0.0
X2=2.0;Y2=0.0
X3=1.0;Y3=-1.0
GO,L.3
)
SET,BREAKPOINT,L.4 (
LIST,VALUES,P.RD
QUIT
)
GO
7/8/9 in column 1
      PROGRAM RD
   10 READ(*,*,END=20,ERR=20) X1,Y1,X2,Y2,X3,Y3
      CALL AREA(X1,Y1,X2,Y2,X3,Y3,A)
      GO TO 10
   20 STOP
      END
      SUBROUTINE AREA(X1,Y1,X2,Y2,X3,Y3,A)
      S1=SQRT((X2-X1)**2 + (Y2-Y1)**2)
      S2=SQRT((X3-X1)**2 + (Y3-Y1)**2)
      S3=SQRT((X3-X2)**2 + (Y3-Y2)**2)
      T=(S1+S2+S3)/2.0
      A=SQRT(T*(T-S1)*(T-S2)*(T-S3))
      RETURN
      END
7/8/9 in column 1
0.0 0.0 2.0 0.0 0.0 2.0
0.0 1.0 0.5 2.0 -1.0 1.2
6.1 2.0 0.1 -4.0 3.2 7.0
6/7/8/9 in column 1
```

Figure D-1. Sample Job Deck for Batch Mode Debugging

```
CYBER INTERACTIVE DEBUG
SET,BREAKPOINT,L.2 [
IN COLLECT MODE
X1=0.0;Y1=0.0
X2=2.0;Y2=0.0
X3=1.0;Y3=-1.0
GO,L.3
]
END COLLECT
SET,BREAKPOINT,L.4 [
IN COLLECT MODE
LIST,VALUES,P.RD
QUIT
]
END COLLECT
GO
X1=0.0;
Y1=0.0
X2=2.0;
Y2=0.0
X3=1.0;
Y3=-1.0
GO,L.3
LIST,VALUES,P.RD
P.RD
A = 1.0,    X1 = 0.0,    X2 = 2.0,    X3 = 1.0,    Y1 = 0.0,    Y2 = 0.0,    Y3 = -1.0
QUIT
```

Figure D-2. Listing of File DBUGOUT

SUMMARY OF CID COMMANDS

Table E-1 summarizes the CYBER Interactive Debug (CID) commands described in this guide, and specifies the page where more detailed information can be obtained.

TABLE E-1. CID COMMAND SUMMARY

Command	Short Form	Described on Page	Description
assignment (var=expr)		3-20	The value of the expression on the right of the equal sign replaces the current value of the variable on the left of the equal sign.
CLEAR,AUXILIARY	CAUX	3-28	Purges the auxiliary output file.
CLEAR,BREAKPOINT	CB	3-7	Removes breakpoints.
CLEAR,GROUP	CG	4-4	Removes command group definitions.
CLEAR,OUTPUT	COUT	3-28	Turns off output to the terminal.
CLEAR,TRAP	CT	3-14	Removes traps.
DISPLAY	D	3-19	Displays the contents of program locations.
EXECUTE	EXEC	4-8	Resumes execution of the user program.
GO		4-8	Resumes execution of the user program or of a suspended command sequence.
IF		4-10	Provides for conditional execution of CID commands.
JUMP		4-12	Causes a transfer of control within a command sequence.
LABEL		4-13	Designates a label to be used as the destination of a JUMP command.
LIST,BREAKPOINT	LB	3-7	Displays information about breakpoints defined for a debug session.
LIST,GROUP	LG	4-4	Displays information about command groups defined for a debug session.
LIST,MAP	LM	3-25	Displays load map information.
LIST,STATUS	LS	3-26	Displays information about the current status of the debug session.
LIST,TRAP	LT	3-14	Displays information about traps defined for a debug session.
LIST,VALUES	LV	3-17	Displays names and values of program variables.
MESSAGE		4-2	Displays a string of characters.
MOVE	M	3-21	Moves data from one location to another.
PAUSE		4-8	Suspends execution of a command sequence.

TABLE E-1. CID COMMAND SUMMARY (Contd)

Command	Short Form	Described on Page	Description
PRINT		3-18	Displays the contents of program variables.
QUIT		2-3	Terminates the debug session.
READ		4-4	Executes a group or file sequence or trap, breakpoint, and group definitions saved on a file.
SAVE,BREAKPOINT	SAVEB	4-14	Writes breakpoint definitions to a file.
SAVE,GROUP	SAVEG	4-14	Writes group definitions to a file.
SAVE,TRAP	SAVET	4-14	Writes trap definitions to a file.
SET,AUXILIARY	SAUX	3-28	Establishes an auxiliary output file.
SET,BREAKPOINT	SB	3-6	Defines a breakpoint.
SET,HOME	SH	3-2	Designates a home program.
SET,OUTPUT	SOUT	3-27	Selects output types to be displayed at the terminal.
SET,GROUP	SG	4-4	Defines a command group.
SET,INTERPRET,ON	SI ON	3-15	Turns on interpret mode.
SET,INTERPRET,OFF	SI OFF	3-15	Turns off interpret mode.
SET,TRAP,LINE	ST L	3-11	Defines a LINE trap.
SET,TRAP,OVERLAY	ST OVL	5-2	Defines an OVERLAY trap.
SET,TRAP,STORE	ST S	3-12	Defines a STORE trap.
SUSPEND		4-15	Suspends the debug session.
TRACEBACK		3-4	Lists a subroutine call sequence.

INDEX

COMMENT SHEET

MANUAL TITLE: CYBER Interactive Debug Version 1 Guide for Users of FORTRAN Version 5

PUBLICATION NO.: 60484100

REVISION: C

This form is not intended to be used as an order blank. Control Data Corporation welcomes your evaluation of this manual. Please indicate any errors, suggested additions or deletions, or general comments on the back (please include page number references).

_____ Please reply   _____ No reply necessary

FOLD   FOLD

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS   PERMIT NO. 8241   MINNEAPOLIS, MINN.

POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION

Publications and Graphics Division
P.O. BOX 3492
Sunnyvale, California 94088-3492

CUT ALONG LINE

FOLD   FOLD

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.
FOLD ON DOTTED LINES AND TAPE

NAME:

COMPANY:

STREET ADDRESS:

CITY/STATE/ZIP:

TAPE   TAPE

CORPORATE HEADQUARTERS, P.O. BOX 0, MINNEAPOLIS, MINN. 55440
SALES OFFICES AND SERVICE CENTERS IN MAJOR CITIES THROUGHOUT THE WORLD

LITHO IN U.S.A.

CONTROL DATA CORPORATION